Enabling dedicated single-cycle connections over a shared Network-on-Chip

by

Tushar Krishna

B.Tech., Indian Institute of Technology Delhi (2007)
M.S.E, Princeton University (2009)

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

Doctor of Philosophy
in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2014

Author . . . .
Department of Electrical Engineering and Computer Science
Oct 25, 2013

Certified by . . . .
Li-Shiuan Peh
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Enabling dedicated single-cycle connections over a shared Network-on-Chip

by

Tushar Krishna

Submitted to the Department of Electrical Engineering and Computer Science
on Oct 25, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Adding multiple processing cores on the same chip has become the de facto design choice as we continue extracting more and more performance/watt from our chips in every technology generation. In this context, the interconnect fabric connecting the cores starts gaining paramount importance. A high latency network can create performance bottlenecks and limit scalability. Thus conventional wisdom forces coherence protocol and software designers to develop techniques to optimize for locality and keep communication to the minimum. This dissertation challenges this conventional wisdom. We show that on-chip networks can be designed to provide extremely low-latencies while handling bursts of high-bandwidth traffic, thus reversing the trade-offs one typically associates with Private vs. Shared caches, or Broadcast vs. Directory protocols.

The dissertation progressively builds a network-on-chip fabric that dynamically creates single-cycle network paths across multiple-hops, for both unicast and collective (1-to-Many and Many-to-1) communication flows. We start with a prototype chip demonstrating single-cycle per-hop traversals over a mesh network-on-chip. This design is then enhanced to support 1-to-Many (multicast) and Many-to-1 (acknowledgement) traffic flows by intelligent forking and aggregation respectively at network routers. Finally, we leverage clock-less repeated wires on the data-path and propose a dynamic cycle-by-cycle network reconfiguration methodology to provide single-cycle traversals across 9-11 hops at GHz frequencies. The network architectures proposed in this thesis provide performance that is within 12% of that provided by an idealized contention-free fully-connected single-cycle network. Going forward, we believe that the ideas proposed in this thesis can pave the way for locality-oblivious shared-memory design.

Thesis Supervisor: Li-Shiuan Peh
Title: Professor of Electrical Engineering and Computer Science

To my family

asato ma sad gamaya

tamaso ma jyotir gamaya

From delusion lead me to truth

From ignorance lead me to knowledge

–Brhadaranyaka Upanisad, I.iii.28

Acknowledgments

I can no other answer make, but, thanks, and thanks.
- William Shakespeare, *Twelfth Night, Act III, Scene III*

My PhD years have been, and am sure will remain, one of the best phases of my life. I have been really lucky to have had the chance to work with and be mentored by Prof Li-Shiuan Peh, my research advisor. Thanks to her, I have had the unique opportunity of spending quality time at Princeton University, MIT and National University of Singapore. I have also had the privilege of presenting our work at conferences around the globe - from USA to Europe to South America to Asia. Li-Shiuan is one of the most fascinating people I have met in my life. Her enthusiasm, excellent insights, strong technical expertise, and systematic problem solving approach will stay with me throughout my professional career. I think she struck a wonderful balance between being a highly indulgent advisor, and yet ensuring that I was making progress and not getting stuck in a lull. But more importantly, I also view her as a role model for her immense energy, optimism and positive attitude towards everything in life. Thank you Li-Shiuan! I look forward to future peer interactions with you.

During the course of the last few years, I have had the honor of interacting and working with some of the most distinguished researchers in our field. A special mention for my thesis committee members Prof Srini Devadas and Prof Joel Emer who have played a major role in shaping my final thesis, and I thank them for their insightful feedback and comments. Interactions with Srini for my thesis, the Angstrom Project, and my Research Qualifying Examination instilled in me the drive to work on big impactful ideas; I thank him for that. I had the good fortune of interacting with Joel, not just for my thesis, but also as a TA for the computer architecture course; I admire Joel's meticulous approach, and I learnt a lot during our discussions trying to chalk out the in-depth design details of the pedagogical architectures for the course. I also had the privilege of working with Prof Arvind for the same computer architecture course, and was fascinated by his love for teaching and attention to detail, and thoroughly enjoyed our conversations. I thank him for looking out for me when I was going through a health issue during that time. I found the TA experience to be one of the high-points of my PhD experience. I thank Prof Anantha Chandrakasan whose circuits course I got an opportunity to take at MIT, and with whom I collaborated for a chip tapeout.

The direction of my PhD thesis was shaped by my internships at AMD Research, where my goal was to push the performance limits of current state-of-the-art networks in order to simplify the design constraints on the coherence and application layer. I thank Steve Reinhardt and Brad Beckmann for guiding me through the initial phases of my research experience. A special thanks also to Prof Patrick Chiang from Oregon State University and Prof Mattan Erez from University of Texas at Austin with whom I collaborated on two projects when I started.

Thank you Maria Rebelo, administrative assistant to Li-Shiuan, for all the help through the years. I would also like to acknowledge Prof Harry Lee - my graduate counselor at MIT, and Janet Fischer - graduate administrator at the EECS office, for helping me through the administrative requirements at MIT.

Six years is a long time, and I have had a chance to interact with and forge lasting friendships with most of the students (alumni and current) in my group. Amit Kumar was the senior-most student when I joined, and my thesis has been built upon his top-notch research. It was a pleasure getting mentored by him, learning from him, and drinking with him during his post-defense party. Niket Agarwal has probably had the biggest influence on me during my PhD life. I thank him for convincing Li-Shiuan to take me as her student in the first place, and being almost like a second advisor - someone I could go to with any problem (technical or non-technical) and come back with an answer. Kostas Aisopos was my closest buddy in the group, and I miss the awesome times we had at Princeton, MIT, Singapore, Bangkok and Brazil. Manos Koukoumidis, I remember working late into the night at Stata Center and having hot pop-corn packets thrown at me from the floor above! Thanks for the pop-corn and your (homemade attempts at) banana bread. Owen Chen has been my best collaborator, critic, and friend through my PhD years. It would be a harder sell getting an idea/design approved by Owen, than it would be by my advisor! I will miss the endless technical discussions and gossip sessions during our prolonged coffee breaks. I thank Sunghyun Park for laying the circuit foundations for the ideas in many of my projects. It has been a pleasure working with you and I look forward to another beer session at Bon Chon! Bhavya Daya, I admire your work ethic and will remember the technical discussions and high-caffeine induced laughs we all had during the SCORPIO chip tapeout. We still need to plan the post-tapeout celebration party! Anirudh Sivaraman, you were a constant source of information (seriously you know more about my field than me!) and entertainment. It has been great hanging out with you at MIT and Singapore. Jason Gao, "you *did* build that". It has been a great pleasure to be part of your research experiments involving us walking and driving, through sun and snow, in Boston and Singapore, carrying iPhones and Galaxy Notes. Woo Cheol Kwon is one of the smartest people I have interacted with. I find it amazing how he can simplify complex-looking ideas to their core and I have always valued his opinion on what can work and what cannot in real systems. It has been a pleasure interacting with you. Pablo Ortiz is probably more mature than me, even though he is years younger than me, and I have enjoyed the witty conversations we have both had as a result of this. Pablo loves to point at my hair turning gray and remind me that graduate school is not the same as *undergrad college life* and I conveniently choose to disagree. Last, but definitely not the least, Suvinay Subramanian. He would like me to believe I have been a mentor to him, but on the contrary I would like to acknowledge him for his clarity of thought and research vision which I really admire. You are a great friend and a social nucleus within the group. Best wishes to you all. To the students currently in the group: I look forward to your graduation.

My close friends in Cambridge have been like a family away from home; they have been with me through the personal ups and downs I faced over the past few years. Thank you to my current and former apartment mates - Siddharth Bhardwaj, Harshad Kasture, Sourav Padhy, Murali Vijayaraghavan, Amit Soni, and Varun Ramanujam - for the entertaining chats on nights and weekends about anything from US foreign policy to Bollywood movies. Thanks Abhinav Agarwal and Murali Vijayaraghavan for the awesome gym sessions. Thanks Hemant Bajpai, Anushree Kamath, Satyam Agrawalla, and Pratibha Pandey for the great dinners and movie outings. Thanks Varun Kumar and Jitendra Kanodia for the fun times both at Princeton and at Cambridge. I have had a special rapport with each one of you personally and I cherish that bond. A shout out to some of my close friends who have shared the graduate school experience with me across various universities, and have been a source of mutual support. Arnab Sinha, Prakash Prabhu, Divjyot Sethi, Kapil Anand and Anjul Patney: great job all of us!

My family's support has been enormous through my graduate school years. Thank you Papa - I miss you. Thank you Amamma (granny) - you have been a constant source of guidance throughout my life, and this thesis is a result of your blessings. And the biggest thank you to Mom - since words cannot do any justice to how much you have done for me, I will just say I am proud to have followed your footsteps and here's to another Dr. Krishna! Cheers.

Contents

List of Figures

List of Tables

1 Introduction

Somewhere, something incredible is waiting to be known.
- Carl Sagan

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."

- Gordon E. Moore, "Cramming more components onto integrated circuits", *Electronics Magazine*, 19 April 1965.

Gordon Moore's observation on the economically viable number of components per integrated circuit is popularly called *Moore's Law*, and continues till today, well beyond the 10 years he initially believed it would last. In the semiconductor industry, this law has become the de facto driver for technological innovation, and has led to a sustained doubling of the number of transistors on a die approximately every 2 years.

Moore's Law, in combination with Dennard's scaling [24] - MOSFET dimensions and operating voltages should be scaled by the same factor to keep electric field constant - allowed each technology generation to produce twice the number of transistors in the same area, with each transistor 1.4× faster than the previous generation at the same power density. This led to an exponential growth in the frequency (perfor-

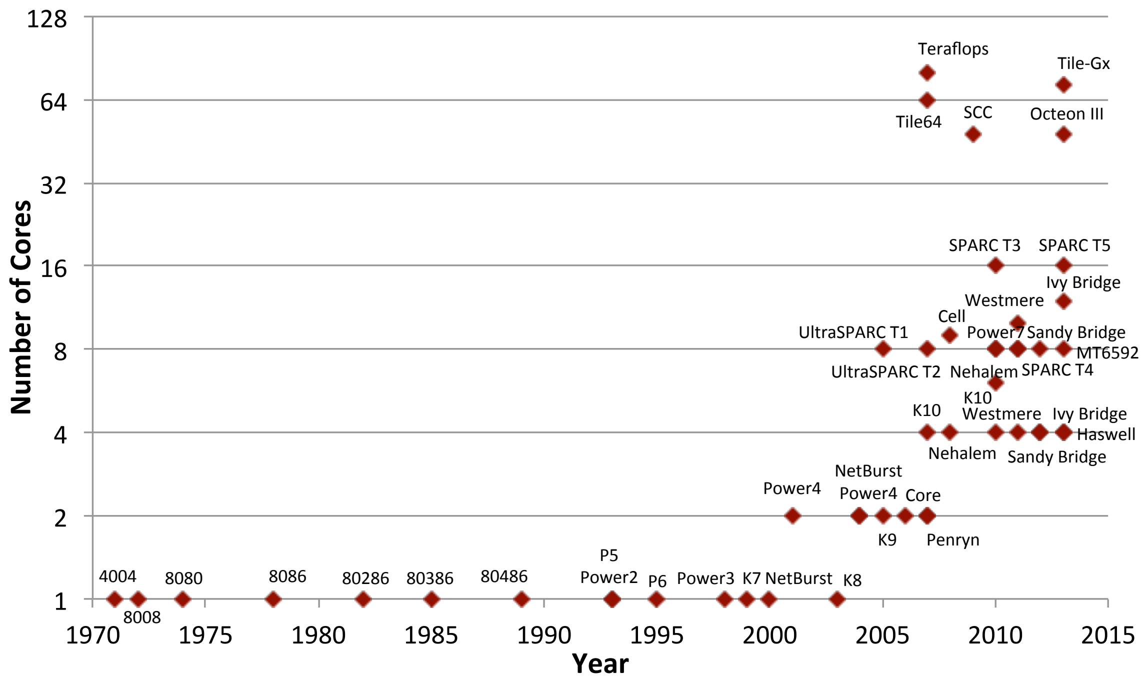


Figure 1-1: Increasing Core Counts over the years.

mance) of chips up to the start of this millennium. But in early 2000s, voltage scaling slowed down because chips were already operating close to the threshold voltage - the physical limit at which transistors turn ON and OFF. The end of voltage scaling also led to the end of frequency scaling to ensure that chips do not cross the power wall ($\sim 100W$) and overheat, as power equals capacitance $\times$ frequency $\times$ voltage-squared. Because of this, and due to ILP (Instruction Level Parallelism) limitations, it was no longer possible to get similar performance gains per unit power as before. Instead, computer architects decided to extract performance by multiplying the number of processing cores on-chip (using the exponentially growing number of transistors from Moore's law) and running them in parallel. This has led to the current wave of Chip Multiprocessors (CMPs) or Multicores.

Figure 1-1 highlights this trend; it plots the number of on-chip cores in several commercial and research CPU architectures from industry leaders like Intel, AMD, IBM, Sun/Oracle, startup Tilera, and others. Over the last decade, we can see that the number of cores has continued to increase. Commercial chips from Intel, AMD and IBM have reached 8-16 cores, while those from Tilera have touched 72 cores.

1.1 Network-on-Chip

In an environment with increasing core counts, the interconnect fabric connecting these cores starts gaining importance. Preferably, each core should have a *dedicated* connection to any core it wishes to communicate with. We call this an *ideal* communication fabric throughout the thesis. However, having dedicated point-to-point wires between all cores would result in a fully-connected topology, blowing up the area beyond a few cores. Instead, for scalability, the solution has been to connect the cores by a network-on-chip (NoC) that they all share. This network has typically been a simple bus or a ring or a crossbar for designs with up to 8-16 cores. However, none of these topologies are very scalable - buses require a centralized arbiter and offer limited bandwidth; rings do not need a centralized arbiter but the maximum latency increases linearly with the number of cores; crossbars offer tremendous bandwidth but are area and power limited. Going forward to tens and hundreds of cores, meshes are making their way into prototype and mainstream designs because of their simplicity, ease of layout, and scalability. Meshes are formed by laying out a grid of wires and adding routers at the intersections which decide who gets to use each wire segment and when, as shown in Figure 1-2. This is known as packet-switching. Routers are necessary to enable the sharing of wires and avoiding collisions, but add delay and power overheads at each hop. In large many-core designs, we run a real risk of limiting our performance and scalability because of the network and its routers.

This dissertation aims at designing a scalable shared network that provides the illusion of *dedicated* wires to most of the messages. In other words, the aim is to design a NoC that mimics the delay/power characteristics of the *ideal* communication fabric.

1.2 On-chip Latency

An on-chip network traversal comprises a series of *hops*[1] via routers from the source core's network interface (NIC) to the destination core's NIC, as shown in Figure 1-2.

[1]A hop refers to the distance between two adjacent tiles.

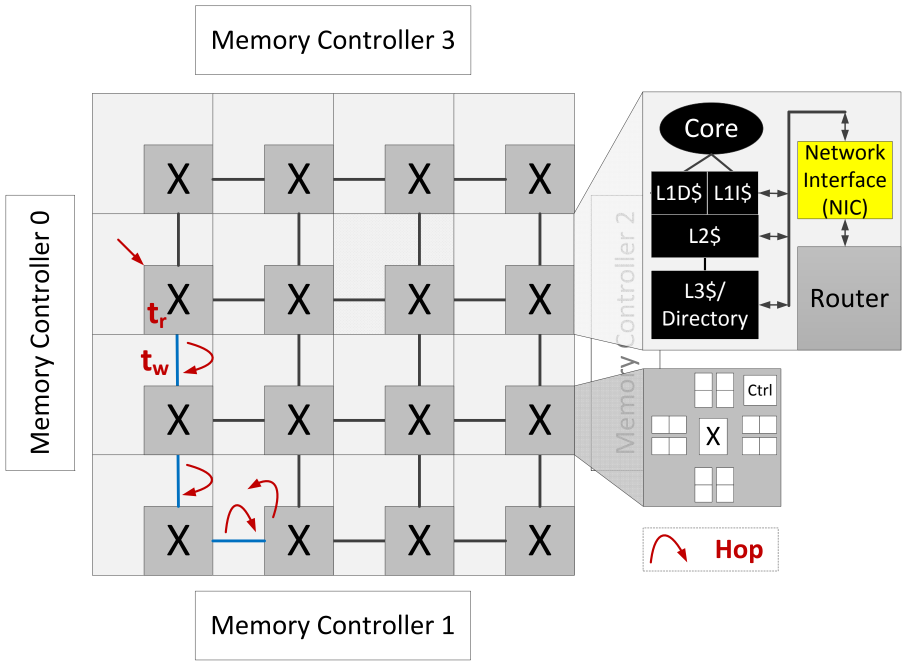


Figure 1-2: Mesh Network-on-Chip.

The latency for any message in the network can be defined as

$$T_N = H \cdot (t_r + t_w) + \sum_{h=1}^{H} t_c(h) \tag{1.1}$$

H is the number of hops; t_r is the router's intrinsic delay; t_w is the wire (between two routers) delay; $t_c(h)$ is the contention delay - number of cycles spent waiting to get access to the switch and output link - at a router h-hops from the start; $\sum_{h=1}^{H} t_c(h)$ is the accumulated contention delay of the network. The first component is fixed, it is defined by the router's microarchitecture. The second component is variable; it depends on the congestion in the network at that moment at various routers, and the efficiency of the routing and flow control mechanisms to handle this congestion. Chapter 2 will discuss more details about these components.

The motivation for this entire thesis stems from Figure 1-3. This figure plots the full-system runtime on a 64-core CMP for a suite of SPLASH-2 and PARSEC applications across three cache coherence protocols: Full-state Directory, AMD HyperTransport, and Token Coherence. The protocols and experimental setup will be

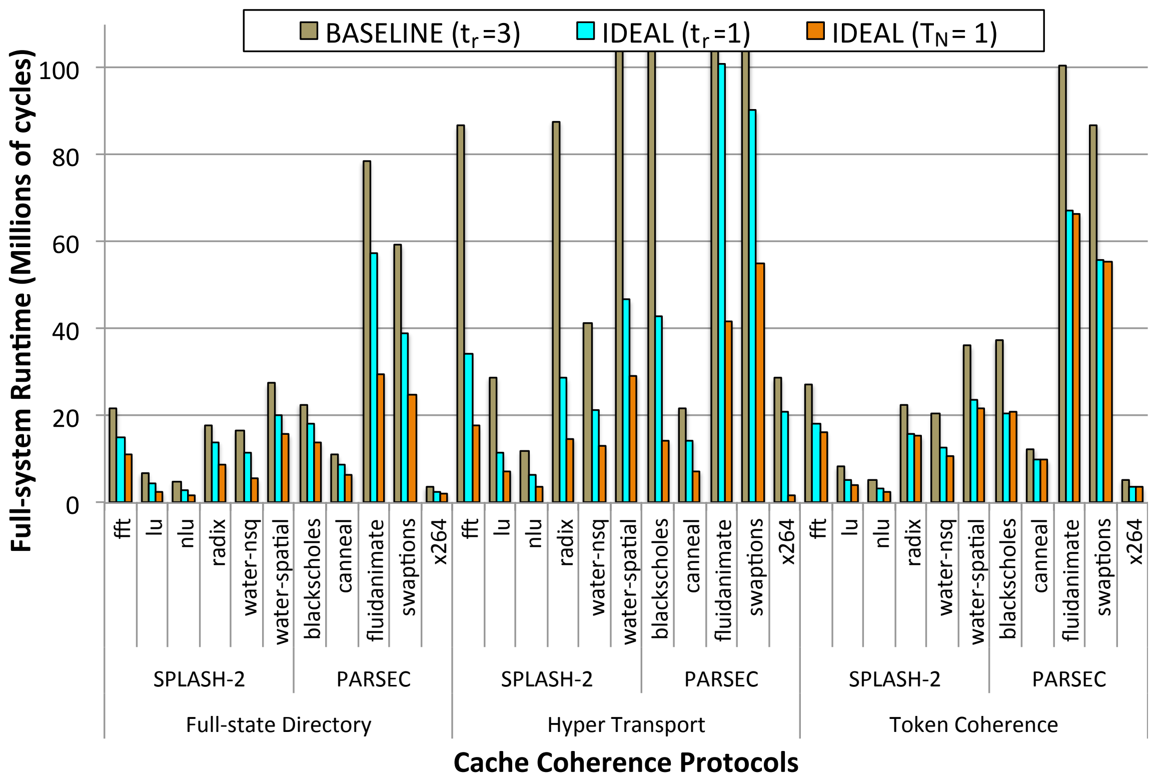


Figure 1-3: Runtime of Applications with different NoC designs on a 64-core CMP.

described in detail in Chapter 2 Section 2.2.1. All simulations are run with three network models:

- **BASELINE (t_r=3)**: This is a network with every router taking 3 cycles, and the link segment between routers taking 1 cycle (i.e., $t_w = 1$), leading to a network delay $T_N = 4 \cdot H + \sum_{h=1}^{H} t_c(h)$. This is similar to Intel's recent 48-core SCC router [38].

- **IDEAL (t_r=1)**: This is an idealized network with 1-cycle routers at each hop, and no contention along the route. The network delay for each message is always $T_N = 2 \cdot H$ (i.e., $t_r = 1$, $t_w = 1$, and $\forall h : t_c(h) = 0$).

- **IDEAL (T_N=1)**: This is an idealized network where the network delay is always 1 cycle, i.e., every message has a dedicated 1 cycle link connecting it to every destination, with no contention (i.e., $H = 1$, $t_r = 0$, $t_w = 1$, and $\forall h : t_c(h) = 0$). We will also refer to this as the ideal *1-cycle network*.

We can see that the IDEAL (t_r=1) NoC lowers runtime by 30-50% across the protocols, and the IDEAL (T_N=1) NoC lowers it by a further 30-50%. The goal of this dissertation is to progressively design a real, practical 1-cycle network. Not only does the ideal 1-cycle network lower runtime, Figure 1-3 shows that it completely changes design trade-offs as well. The BASELINE (t_r=3) NoC suggests that the full-state directory protocol is the most obvious design choice, but with a IDEAL (T_N=1) NoC, it is no longer that clear since the network is no longer the bottleneck. In other words, a NoC that is designed to be extremely low-latency, and efficiently handle the bandwidth requirements of different traffic flows, can in turn ease the design of the coherence protocol and software (OS/compiler) running atop it by relaxing the traditional requirement for locality-aware optimizations. Thus, this dissertation lays the foundation for locality-oblivious shared memory design.

1.3 Dissertation Contributions and Structure

The rest of the dissertation is organized as follows:

- Chapter 2 presents relevant background on NoCs and cache coherence protocols. It also presents our evaluation methodology and performance metrics.
- Chapter 3 describes the detailed router microarchitecture of a 1-cycle router prototype called the SWIFT NoC [53, 69]. This chapter addresses t_r in Equation 1.1, reducing it to 1. This design serves as the baseline, BASELINE (t_r=1), for the rest of the thesis. SWIFT lowers runtime by 26% compared to BASELINE (t_r=3), and is less than 2% away from the IDEAL ($t_r = 1$).
- Chapter 4 presents a collection of routing, flow control and microarchitectural innovations called FANOUT [52, 66] to distribute 1-to-Many traffic flows at single-cycle per-hop within the network. This chapter addresses both t_r and $t_c(h)$ in Equation 1.1 for multicast traffic. For a broadcast-intensive protocol, FANOUT lowers runtime by 15% compared to BASELINE (t_r=1), and is within 2.5% away from the IDEAL ($t_r = 1$).

- Chapter 5 presents a collection of routing, flow control and microarchitectural innovations called FANIN [52] to aggregate/reduce Many-to-1 traffic flows at single-cycle per-hop within the network. This chapter addresses both t_r and $t_c(h)$ in Equation 1.1 for acknowledgement traffic. For an acknowledgement-intensive protocol, FANIN lowers runtime by 9% compared to FANOUT, which is within 1% of the IDEAL ($t_r = 1$).

- Chapter 6 presents a NoC called SMART [50] that can support single-cycle multi-hop traversals, potentially all the way from the source to the destination, i.e., a practical realization of a 1-cycle network. This chapter addresses H in Equation 1.1, breaking the dependence of latency on number of hops. Compared to the BASELINE (t_r=1) NoC, SMART lowers runtime by 27/52% for Private/Shared L2 designs, which is only 9% off the runtime with an IDEAL ($T_N = 1$) NoC. For coherence protocols with 1-to-Many and Many-to-1 flows, SMART reduces runtime by 15-19% on average, which is within 12% of the IDEAL ($T_N = 1$) design.

- Chapter 7 concludes and discusses future research directions.

- Appendix A performs a design space exploration of repeated wires. We observe that repeated wires can go 13-19mm within 1ns and can be leveraged to send signals multiple-hops within a single-cycle on a chip. This study drives the SMART NoC design in Chapter 6.

2 Background

If I have seen further, it is by standing on the shoulders of giants.
- Isaac Newton

This chapter introduces readers to a Network-on-Chip, and discusses its components in sufficient detail for understanding the thesis. Necessary background about both synthetic traffic and cache coherence protocol traffic used to drive the NoC is also presented. We also describe our evaluation methodology and performance metrics.

2.1 Network-on-Chip (NoC) basics

A collection of cores/IPs on the same die necessitates an interconnection between them. We define a stream of communication between any two cores as a communication **flow**. If traffic between cores is deterministic, the interconnect can be tailored to match the communication flow pattern. This is often the case in the SoC (embedded systems) domain. However, with general purpose cores in a shared memory CMP domain, any core could potentially talk to any other core. Having dedicated all-to-all connections is not practical beyond a few cores, and thus the accepted solution has been to use a *shared* communication medium, and perform multiplexing of different flows over it in different cycles. This shared interconnect fabric has been referred to by multiple names in literature: network-on-chip, on-chip network, on-die network,

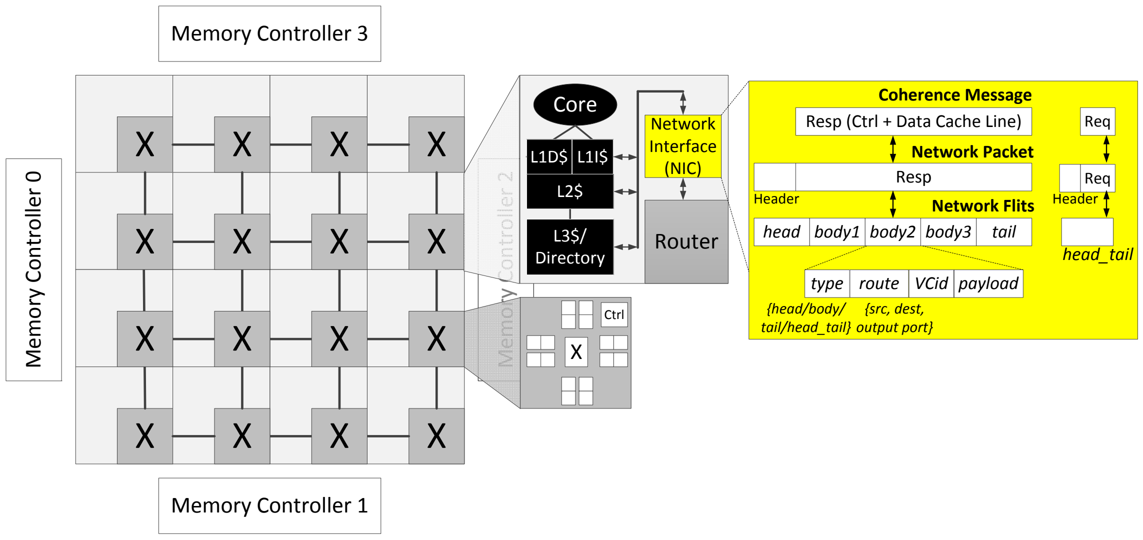


Figure 2-1: Mesh Network-on-Chip Overview.

interconnection network[1] and so on. We use these terms interchangeably in this text.

We assume a tiled CMP design, with each tile comprising a core, a L1 I&D cache, a L2 cache[2], a Network Interface and a Router. This is shown in Figure 2-1. The role of the **Network Interface (NIC)** is to encode L1/L2/Directory coherence requests/responses at the injecting source node into network **packets**, and then break the packet further into **flits**, or flow control units, as shown in Figure 2-1. Flits representing single-flit packets are of the type *head_tail*. Flits within multi-flit packets can be of 3 types: *head*, *body*, *tail*. The role of single vs. multi-flit packets in cache coherence will be discussed later in Section 2.2.1. Within the network, resources (router buffers and links between routers etc) are allocated at the flit granularity. At the destination NIC, flits of a packet are re-assembled back into a packet, and the payload is sent to the appropriate cache controller.

The key features that characterize a NoC are: topology, routing algorithm, flow control, and router microarchitecture. We describe each of these briefly. A more comprehensive description can be found in any book on interconnection networks [22, 25, 40]. A NoC is analogous to a conventional road network, with the roads representing

[1]The term interconnection network spans both off-chip and on-chip networks.

[2]The L2 cache also carries directory state for full-state directory protocols, which will be discussed later in Section 2.2.1.

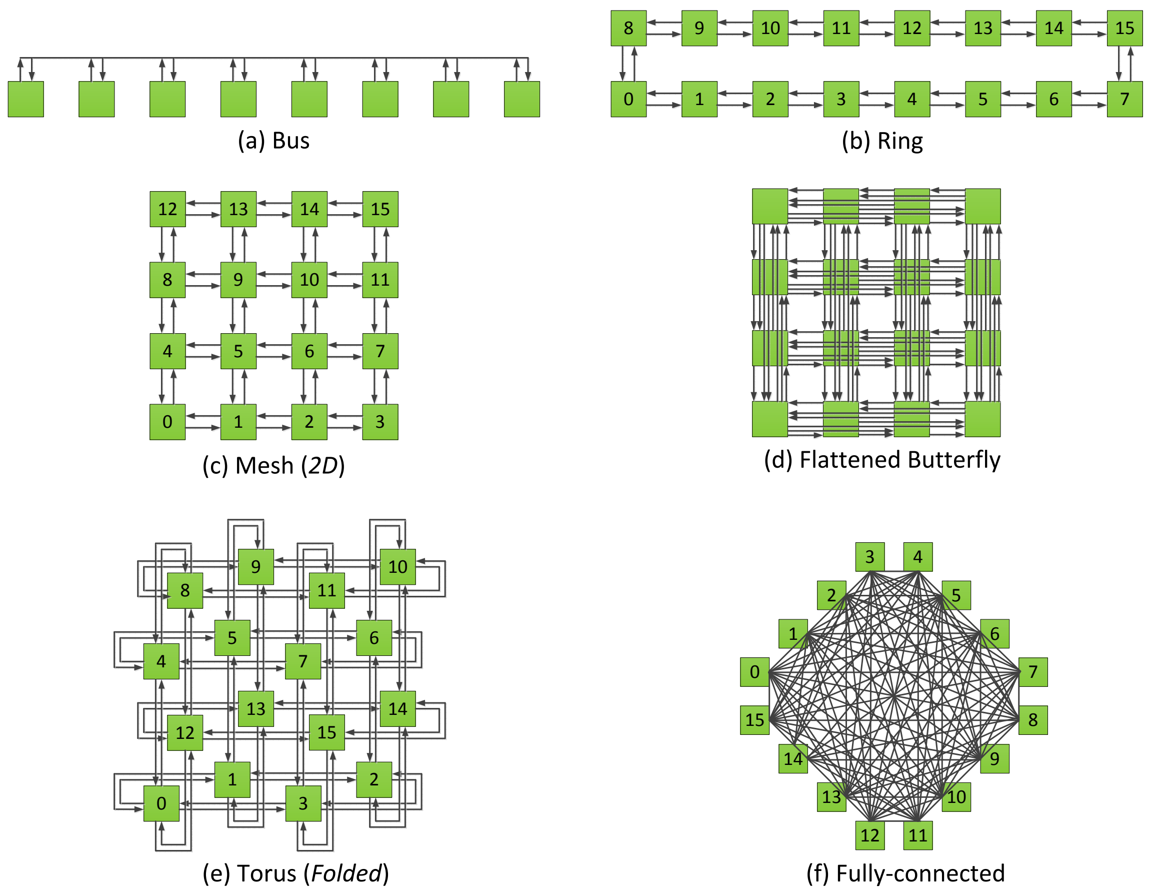


Figure 2-2: Common Network-on-Chip Topologies.

links, traffic signals representing routers, and vehicles representing network packets. We shall use this analogy throughout this section for clarity.

2.1.1 Topology

The topology describes the connection of routers (i.e., tiles) via links/channels, and is equivalent to the layout of the road network to connect various geographic locations. Some common topologies such as Bus, Ring, Mesh, Flattened Butterfly, Torus and Fully-connected are shown in Figure 2-2. We define **hop** to refer to the physical connection between neighboring tiles/routers.

The topology defines the minimum number of hops between a source and destination pair. In a 1D topology, such as a ring, the number of hops increases linearly with number of nodes; in 2D topologies, such as a Mesh and Torus, the number of

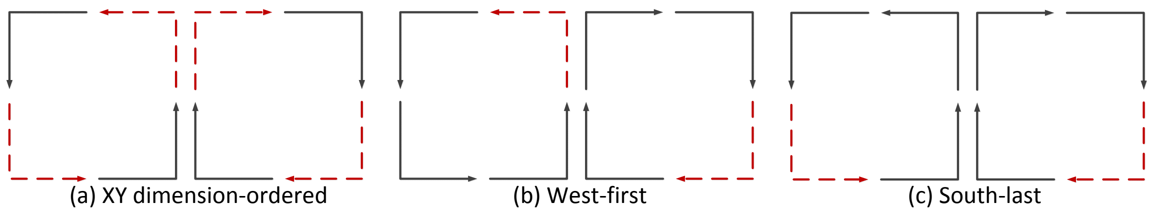


Figure 2-3: Routing Turn Model to avoid deadlocks.

hops increases as a square root of the number of nodes. A Fully-connected topology, while impractical due to layout issues, always offers a 1-hop traversal.

The topology also determines the path diversity available to the flits, i.e., the number of alternate *shortest* paths to get from a source to a destination. For instance, to get from Node-1 to Node-10, there is one path in the Ring ($1 \to 0 \to 8 \to 9 \to 10$), three in the Mesh ($1 \to 2 \to 6 \to 10, 1 \to 5 \to 6 \to 10, 1 \to 5 \to 9 \to 10$) and six in the Torus ($1 \to 2 \to 6 \to 10, 1 \to 5 \to 6 \to 10, 1 \to 5 \to 9 \to 10, 1 \to 2 \to 14 \to 10, 1 \to 13 \to 9 \to 10, 1 \to 13 \to 14 \to 10$).

2.1.2 Routing Algorithm

The routing algorithm determines the series of links (roads) that the flit (vehicle) should take to get from a source node S to a destination node D. Routing algorithms can be minimal or non-minimal. Minimal routes refer to any of the shortest (i.e., minimum possible hop count) paths from S to D. At every hop on a minimal route, the flit moves closer to D. Non-minimal routes on the other hand allow flits to get misrouted and travel away from D before finally turning back and reaching D. Non-minimal routes can help route around hot-spot routers, but will not be considered in this thesis due to their additional traversal delay/energy, and complexity to avoid deadlocks and livelocks.

A further classification of routing algorithms is into deterministic, oblivious and adaptive. A deterministic route is one where the same set of links is always used to get from a particular source S to a particular destination D. The most common deterministic routing scheme in a Mesh, which we use extensively in this thesis, is called **XY dimension-ordered routing**, where a flit always traverses the X direction (i.e., West or East) first, and then turns towards Y (i.e., North or South).Oblivious and adaptive routing schemes allow different flits from S to D to traverse different routes. The only difference between these schemes is that adaptive schemes monitor network congestion and use that as a metric to choose different routes, while oblivious schemes use other metrics (such as theoretical link utilization, or randomness). To avoid deadlocks, all routing schemes can conform to a *turn model* [29] which disallows certain turns in order to avoid creating a cyclic resource dependency. Figure 2-3 shows the turn models for three deadlock-free routing algorithms (XY dimension-ordered, West-first and South-last) with the disallowed turns highlighted.

2.1.3 Flow Control

The flow control determines when a flit can traverse the next link on its route, and when it has to wait at some router. It is equivalent to the operation of traffic signals at a junction. The role of an efficient flow control mechanism is to minimize the latencies at low-loads, and maximize the throughput at high-loads. Both these goals can be achieved by ensuring that network resources (buffers and links) are not idle when there are flits waiting to use them. While the topology and routing algorithm fix the theoretical latency and throughput characteristics for a particular traffic pattern, it is the flow control that determines how close to this theoretical capacity can the the network operate.

We define the latency T_P, in cycles, of a packet in the network as follows:

$$\begin{aligned} T_P &= T_N + T_s \\ &= (H \cdot (t_r + t_w) + \sum_{h=1}^{H} t_c(h)) + (\lceil \frac{L}{b} \rceil - 1) \end{aligned} \tag{2.1}$$

T_N is the network delay of a single-flit, defined in Equation 1.1 in Chapter 1[3]; T_s is the serialization delay incurred by multi-flit packets in a pipelined traversal, and is

[3] H is the number of hops; t_r is the router delay; t_w is the wire (between two routers) delay; $t_c(h)$ is the contention delay - number of cycles spent waiting to get access to the switch and output link - at a router h-hops from the start.

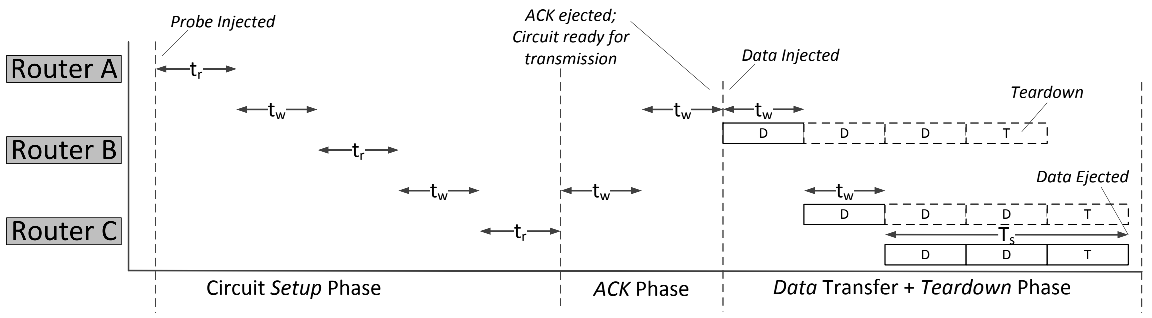


Figure 2-4: Circuit-switching flow control timeline for a message going from Router 1 to Router 3, assuming no contention. Messages incur only wire delay, but each transmission requires a setup and acknowledgement before transmission can begin. (D: Data, T: Teardown).

set by the number of cycles it takes for a packet of length L to cross a channel with bandwidth b (i.e., the number of flits in the packet).

Circuit-switching

Circuit-switching originates from the telephone world, where a dedicated connection is setup between two phones, and lasts throughout the duration of the call. In a circuit-switched NoC, a probe is first sent out from the source to the destination, to *setup* the circuit, i.e., reserve all links along a route for a particular flow. Once the probe reaches the destination, an *acknowledgement* is sent back to the source on the reserved path. The source can now send *data* to the destination at the lowest possible latency (i.e., wire delay) and incurs no buffering or contention at the routers. Once data transfer is complete, a *teardown* message is sent to the destination to free the reserved links and allow them to be reserved by other flows. Figure 2-4 shows all these phases, along with the incurred latency, in a time-space diagram.

During the data transmission phase, circuit switching works like an *ideal* communication fabric, as defined earlier in Chapter 1, with messages incurring only wire delay. However, there should be significant data transmission to amortize the delay in the setup, acknowledgement and teardown phases. Moreover, if data transfer on the reserved circuit is bursty and not sustained, the links will be under-utilized, reducing the realized network throughput.

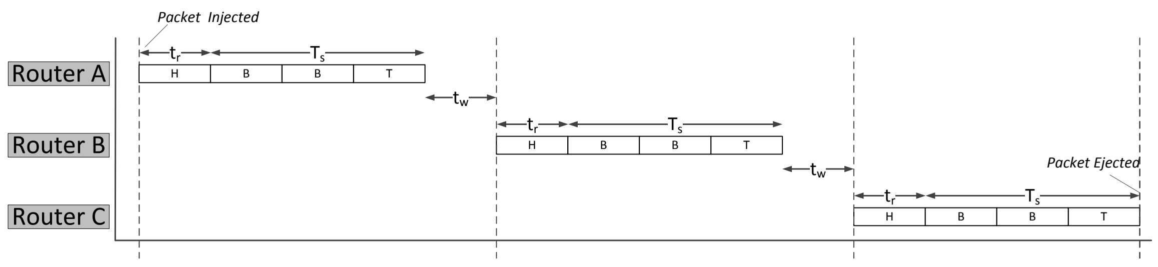


Figure 2-5: Store-and-Forward flow control timeline for a packet going from Router 1 to Router 3, assuming no contention. Transmission occurs at packet granularity. Packets incur serialization delay at each hop. (H: Head flit, B: Body, T: Tail.)

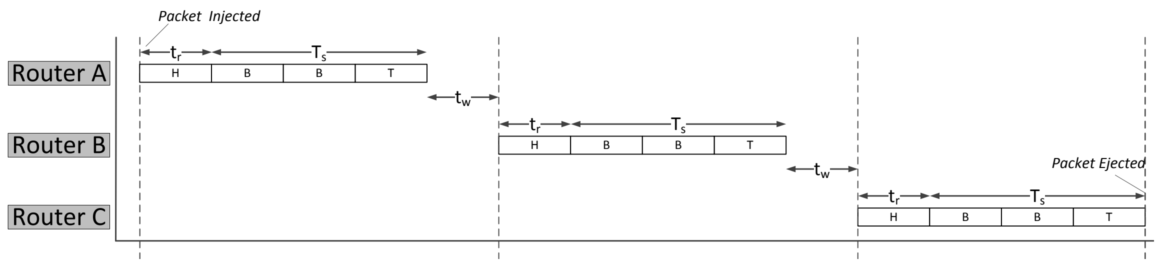


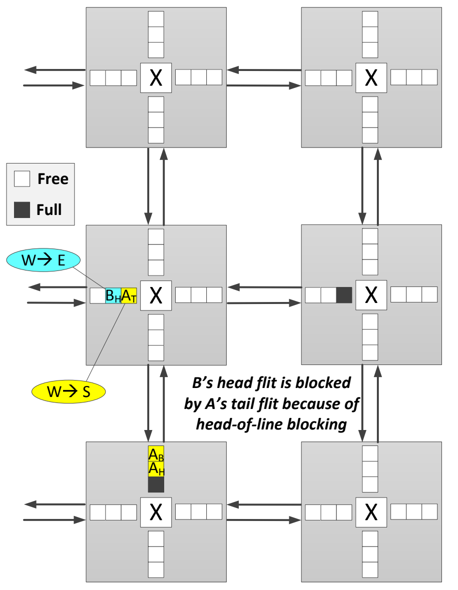
Figure 2-6: Virtual Cut-Through and Wormhole flow control timeline for a packet going from Router 1 to Router 3, assuming no contention. The head flit cuts through the router as soon as it arrives. Packets incur serialization delay only at the last hop. (H: Head flit, B: Body, T: Tail.)

Packet-switching

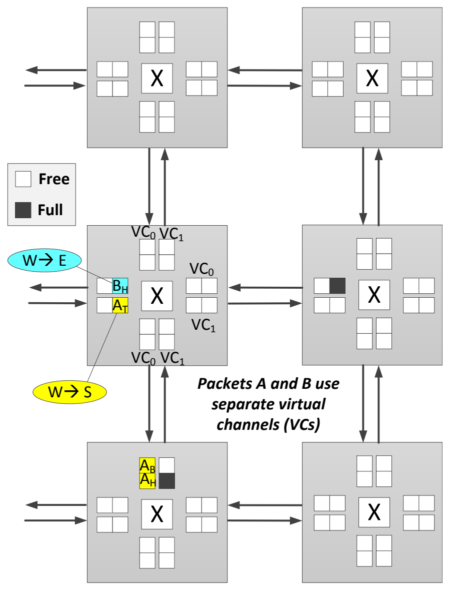
Packet-switching facilitates the time-multiplexing of packets from different flows on the same links. This improves channel utilization and thus boosts throughput. Instead of sending out a setup probe, the source directly injects data packets into the network. These packets allocate links in a hop-by-hop manner all the way till the destination, and wait at intermediate routers in case of contention. Thus there is no setup, acknowledgement or teardown phase. Most interconnection networks today use packet-switched flow control techniques to efficiently allocate the limited buffers and links among contenting flows.

Packet-switched flow control techniques can broadly be classified into the following categories, based on the granularity at which buffers and links are allocated.

In **store-and-forward** flow control, each router waits until an entire packet has been received, before forwarding it to the next router. Buffer space and link band-


Free
Full
W→ E
W→ S
B_H A_T
X
B's head flit is blocked by A's tail flit because of head-of-line blocking
A_B
A_H

(a) **Head-of-Line blocking in wormhole flow control.**



(b) **Virtual Channel (VC) flow control.**

Figure 2-7: Wormhole vs. Virtual Channel flow control.

width are thus allocated at a packet granularity. Figure 2-5 shows the latency incurred by a packet in this scheme.

Virtual cut-through flow control removes the per-hop serialization delay from store-and-forward by allowing the transmission of flits of a packet to begin before the entire packet is received, as shown in Figure 2-6. Storage and bandwidth are still allocated at the packet granularity.

Wormhole flow control reduces the storage overhead of packet-granularity schemes by enabling buffer storage and channel bandwidth to be allocated at flit granularity. While the traversal time-space diagram looks identical to that for Virtual cut-through, routers can now have buffer storage that is smaller than the size of packets, thus reducing router area and power. A key issue with wormhole flow control is that packets go out serially from a router's input port, in the order in which they came in, since there is only one FIFO queue for the physical input channel. If the flit of a packet at the head of the queue gets blocked due to insufficient buffer space due to congestion at its next router, the packet behind it also gets blocked even though it might want to use a separate non-congested route. This is known as **head-of-line blocking**. This is shown in Figure 2-7(a), where the head flit of packet B that wants to go East gets blocked by the tail flit of packet A that wants to go South. Thus the East links go idle even though there is a flit that wants to use them, lowering network throughput.

Another potential issue with wormhole flow control is that a cyclic dependency can be created between the queues at router input ports (if the routing protocol allows all possible turns), leading to deadlocks.

Virtual Channel (VC) flow control removes the head-of-line blocking problem by associating separate queues for different flows at a router, rather than queuing them one behind the other like wormhole routers, even though there is only one physical input/output channel. Virtual Channels are analogous to separate turning and straight lanes at traffic intersections, to prevent turning vehicles getting blocked by vehicles going straight. Figure 2-7(b) highlights how VCs solve the problem incurred by wormhole routers in the previous example. A *head* flit allocates a VC, and arbitrates for the output physical channel bandwidth before it can proceed to the next router. The *body* and *tail* flits use the same VC, but still need to compete for the channel bandwidth with flits in other VCs. A VC is freed once the *tail* flit leaves. We do not allow more than one packet to occupy a VC. When a packet in some VC gets blocked, packets behind it can still traverse the physical channel using other virtual channels, thus solving the head-of-line blocking problem and enhancing throughput.

VCs can also serve a dual role by breaking deadlocks in the network. Even if the routing protocol allows all turns, careful allocation of VCs when making turns can ensure that there is no cyclic resource dependency.

Buffer Management

We assume that flits cannot be dropped in the NoC. This means that an upstream router can send flits to its downstream (neighboring) router only if there is a guaranteed buffer. There are two common ways of communicating buffer availability - credits and on-off.

In **credit** signaling, each upstream router maintains a count of the number of free buffers at its adjacent downstream router. It decrements the count each time a flit is

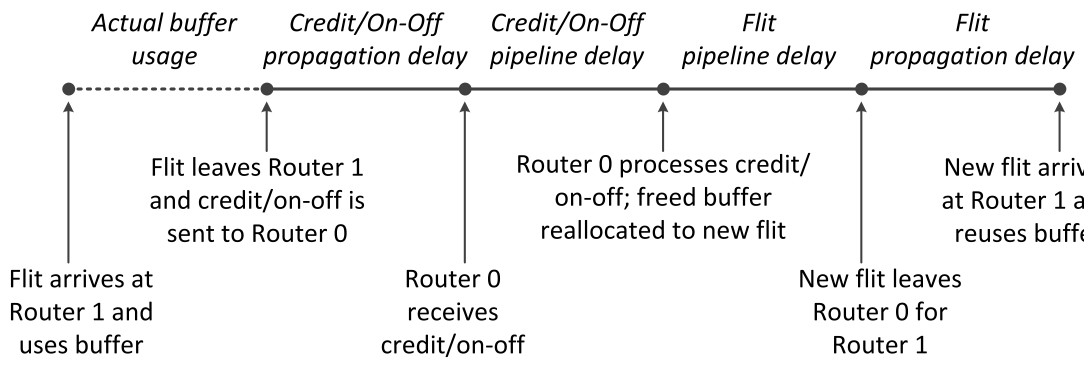

Actual buffer usage
Credit/On-Off propagation delay
Credit/On-Off pipeline delay
Flit pipeline delay
Flit propagation delay
Flit leaves Router 1 and credit/on-off is sent to Router 0
Router 0 processes credit/ on-off; freed buffer reallocated to new flit
New flit arrives at Router 1 and reuses buffer
Flit arrives at Router 1 and uses buffer
Router 0 receives credit/on-off
New flit leaves Router 0 for Router 1

Figure 2-8: Buffer Turnaround Time [40].

sent out. When a flit leaves the downstream router, it sends a credit bit back to the upstream router which increments its credit count. For VC flow control, the credit count is maintained on a per-downstream-VC basis, and the credit signal carries the credit bit, the VCid, and an additional bit to indicate if the VC is now free or not.

In **on-off** signaling, the downstream router sets a bit high (low) if the number of free buffers is above (below) some threshold value. The upstream router sends a flit only if the on-off bit is high. The threshold value is set by the **buffer turnaround time**. This is the round-trip delay (in cycles) for the on-off signal to go to the upstream router, be processed and be visible to the arbitrating flits. The threshold value guarantees that all flits received in between the time that the bit is turned low and the upstream router stops sending flits have a free buffer available. For VC flow control, an on-off bit is required for every VC. On-off signaling can end up lowering throughput compared to credit-based signaling since the on-off signal could be low and yet there could be idle buffers at the next router.

The buffer turnaround time determines the minimum number of VCs and/or buffers-per-VC to avoid self-throttling of the system. It depends on the wire propagation delay and the router pipeline depth, as shown in Figure 2-8. The longer the delay, the longer is the idle time for a free buffer, the lower is the buffer utilization and poorer is the throughput.

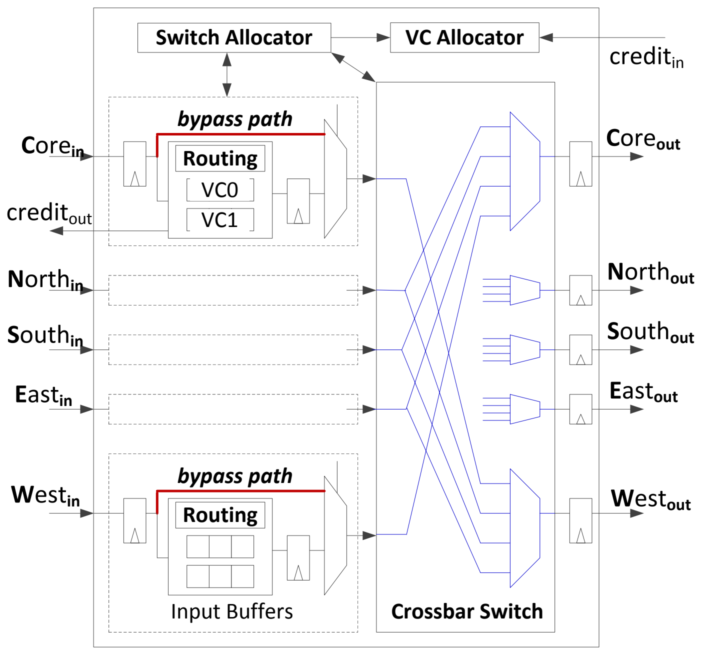


Figure 2-9: Microarchitecture of a 5-port Mesh Router.

2.1.4 Router Microarchitecture

The microarchitecture of the router comprises the logic and state blocks that implement the components described so far. In the traffic signal analogy, the microarchitecture is similar to the design of the traffic intersection, such as the different lanes (left-only, right-only etc), the algorithm running inside the signal to decide when to switch from Red to Green, and so on.

Figure 2-9 shows the microarchitecture of a state-of-the-art NoC router. We show a 5-ported router (for a mesh).

Each input port has buffers that are organized into separate VCs. Buffers are FIFO queues that can be implemented using Flip Flops or register files or SRAM.

Each input port connects to a crossbar switch which provides cycle by cycle non-blocking connectivity from any input port to any output port. A crossbar is fundamentally a mux at every output port. Mux-based crossbars are actually implemented by synthesizing muxes at every output port, while matrix crossbars layout the crossbar as a grid with switching elements at cross-points.

Each input port also houses a route compute unit, an arbiter for the crossbar's input port, and a table tracking the state of each VC. Each output port has an arbiter for the crossbar's output port, and also tracks the free VCs and credits at the neighboring router's input port. A $n : 1$ arbiter allows up to n requests for a resource, and grants it to one of them. Matrix arbiters [22] maintain fairness across cycles and are used in this thesis.

Each flit that goes through a router needs to perform the following actions on its *control-path*:

- **Route Compute (RC).** All *head* and *head_tail* flits need to compute their output ports, before they can arbitrate for the crossbar. RC can be performed either by a table lookup, or simply by combinational logic. The former is used for complex routing algorithms, while the latter is used for simpler routing schemes like XY which we assume in most of this thesis. To remove RC from the critical path, we use lookahead routing [27] where each flit computes the output port at the next router, instead of the current one so that its output port request is ready as soon as it arrives.

- **Switch Allocation (SA).** All flits arbitrate for access to the crossbar's input and output ports. For a $n{\times}n$ router with v VCs per input port, Switch Allocation is fundamentally a matching problem between n resources (output links) and $n{\times}v$ contenders (total VCs in the router). To simplify the allocator design in order for it to be realizable at a reasonable clock frequency, we often use a separable allocator [67]. The idea is to first arbitrate among the input VCs at each input port using a $v : 1$ arbiter at every input port, and then arbitrate among the input ports using a $n : 1$ arbiter at every output port[4]. We call these stages **SA-i** and **SA-o** respectively in this thesis.

- **VC Allocation (VA).** All flits need a guaranteed VC at the next router before proceeding. VC Allocation is only performed by *head_tail* and *head* flits, while *body* and *tail* flits use the same VC as their *head*. VC Allocation can also be

[4]Since we disallow u-turns in minimal routing schemes, the arbiter at the output ports can be $n - 1 : 1$.

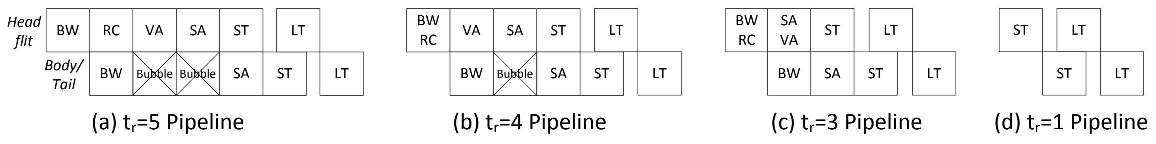


Figure 2-10: Evolution of Router Pipelines.

performed in a separable manner [67] like SA. In this thesis, we use a simpler VA scheme proposed by Kumar *et al.* [55] which we refer to as **VC Select (VS)**. Each output port maintains a queue of VC ids corresponding to the free VCs at the neighbor's input port. The SA winner for that output port gets assigned the VCid at the head of the queue, and the VCid is dequeued. When a VC becomes free at the next router and it sends back a credit, the VCid is enqueued into the queue. If the free VC queue is empty, then flits are not allowed to perform SA.

Once a flit completes RC, SA and VA, it can proceed to its *data-path*:

- **Switch Traversal (ST).** Winners of SA traverse the crossbar in this stage. The select lines of the crossbar are set by the grant signals of SA.
- **Link Traversal (LT).** Flits coming out of the crossbar traverse the link to the next router.
- **Buffer Write (BW).** Incoming flits are buffered in their VC. While the flit remains buffered, its control-path (RC, SA and VA) is active.
- **Buffer Read (BR).** Winners of SA are read out of their buffers and sent to the crossbar.

Router Pipeline

Early on-chip router prototypes [22, 37] were modeled similar to off-chip routers. Their pipeline is shown in Figure 2-10(a). This design has a 5-stage router, i.e., $t_r = 5$. Lookahead routing [27], which computes the route one hop in advance, shortens the router pipeline by one stage, as shown in Figure 2-10(b), allowing VA and SA to commence as soon as the route is read out in the first stage. Speculative VC allocation [67] or VC Select allow VA to occur in parallel to SA, reducing the pipeline even further to 3-cycles, as shown in Figure 2-10(c). To this 3-stage baseline router, which is similar to Intel's recent 48-core SCC router [38], recent research has proposed speculative pre-arbitration of the crossbar to reduce the pipeline to 1-cycle within the router, as shown in Figure 2-10(d). If the pre-arbitration (i.e., VA and SA) succeeds, the crossbar is setup for the incoming flit to directly traverse it, bypassing the conventional BW stage. If the pre-arbitration fails, the incoming flit is buffered as before and continues to arbitrate for the switch and VC. This design was fabricated as part of this thesis work, and will be described in detail in Chapter 3. It will be referred to as **BASELINE (t_r=1)** throughout the thesis.

2.2 Traffic through the on-chip interconnect

In shared memory systems, the on-chip network interconnects the memory subsystem (L1, L2, directory, memory controller etc). The traffic through the network is thus cache coherence traffic. In addition we stress test our network with myriad synthetic traffic patterns to characterize the latency/throughput characteristics. Both these kinds of traffic domains are described in this section.

2.2.1 Cache Coherence Protocols

The role of the cache coherence protocol is to maintain the semantics of one writer or many readers in parallel programs. We assume a CMP design, with a private L1 per tile, and a shared L2 distributed across all tiles, as was shown earlier in Figure 2-1. Each L2 acts as a home node for part of the address space, and holds the directory state for each cache line.

We classify cache coherence protocols into 4 categories.

- **Full-state Directory.** In this design, the directory has a bit-vector to track all sharers and the owner (if any) for any line. The storage requirement for this

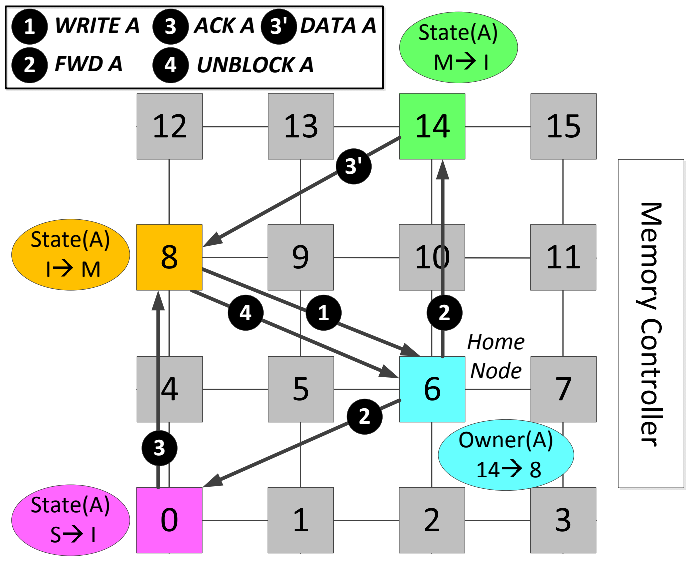


Figure 2-11: Cache Coherence transactions with full-state directory.
1. The requester (#8) sends a write miss to the home node (#6).
2. The home node forwards it to the owner (#14) and the sharer (#0).
3. The owner invalidates its copy and sends the data to the requester. The sharer invalidates its copy and sends an ACK to the requester.
4. The requester then unblocks the home node.

design goes up as O(N) where N is the number of cores.

Figure 2-11 shows a transaction in a full-state directory protocol, and the corresponding messages in the network. L2 #8 sends a write miss (*WRITE A*) to its home node L2 #6 which houses a part of the distributed directory. L2 #6 forwards (*FWD A*) this request to the sharer (S) of this data L2#0 and the owner (O) of this data[5] L2# 14. L2 #14 responds with data (*DATA A*) to L2 #8, while L2#0 sends an invalidation acknowledgement (ACK) to L2 #8. On receiving all ACKs and data, L2 #8 then unblocks its home node (*UNBLOCK A*). At the end of the transaction, L2 #8 holds line A in modified (M) state, while L2# 14 and L2# 0 hold the line in invalid (I) state.

- **Partial-state Directory.** In these designs, the directory only tracks the owner [21] or a subset of sharers [58] or tracks lines at a coarser granularity [34], to reduce the storage requirement. To provide coverage over the complete chip,

[5]If there is no owner on-chip, L2 #6 also forwards this request to the memory controller since DRAM is the owner of the line.

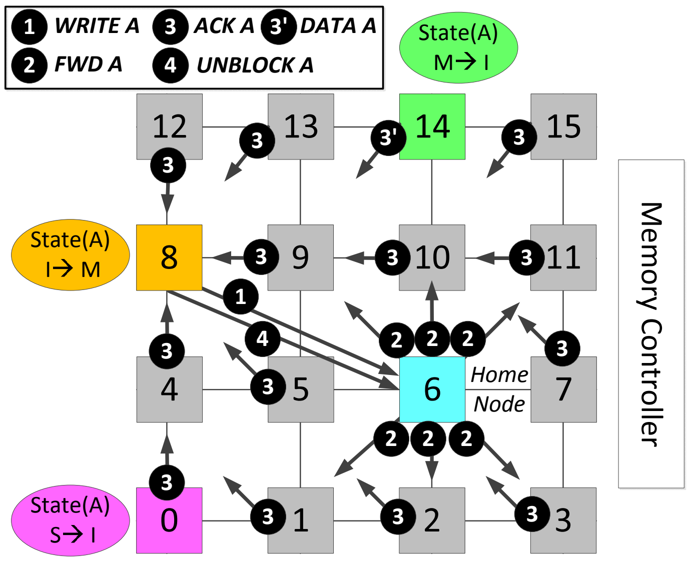


Figure 2-12: Cache Coherence transactions with HyperTransport (broadcast protocol with partial/no-state directory).
1. The requester (#8) sends a write miss to the home node (#6).
2. The home node broadcasts it to all nodes (1-to-all flow).
3. The owner (#14) invalidates its copy and sends the data to the requester. The sharer (#0) invalidates its copy. All nodes (sharers and non-sharers) send an ACK to the requester (all-to-1 flow).
4. Upon receiving all ACKs, the requester unblocks the home node.

it resorts to occasional broadcasts.

For the same coherence transaction as before, Figure 2-12 shows the messages sent by a partial-state directory. Upon receiving the write miss from L2#8, the home node (L2 #6) performs a full-chip broadcast to invalidate the sharers. L2 #14 responds with data (*DATA A*) to L2 #8. All other nodes (both sharers and non-sharers) send an invalidation ACK to L2 #8. Upon receiving ACKs (and data) from all nodes on-chip, L2#8 unblocks the home node. If there is no owner on-chip, then the memory controller, which also received the broadcast, responds with data.

A read miss, on the other hand, is served by a unicast being sent from the home node to the tracked owner L2# 14.

- **No-state Directory.** In this design, the directory does not have any state. The home node in Figure 2-12 needs to perform a full-chip broadcast for both

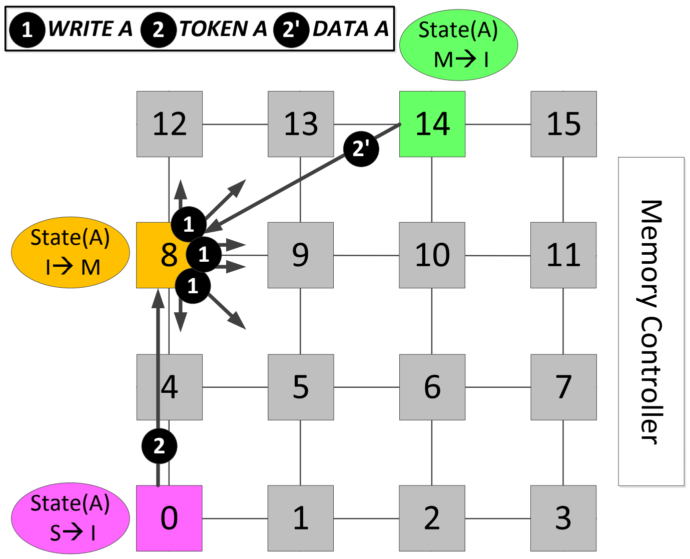


Figure 2-13: Cache Coherence transactions with Token Coherence (broadcast protocol with no directory).
1. The requester (#8) broadcasts a write miss over the chip.
2. The owner (#14) invalidates its copy and sends the data and all tokens to the requester. The sharer (#0) invalidates its copy and sends its token to the requester.

a read and write miss, and all L2's need to respond to the requester with an ACK (or data if an L2 is the Owner), before it can unblock the home node. The role of the "directory" in this design is simply to provide an ordering point among different requests to the same line, to help maintain memory consistency semantics [6]. AMD's HyperTransport^{TM} [9], used in its early Opteron chips, is an example of a no-state directory.

The actions in **HyperTransport (HT)**, which we run for this thesis, are similar to Figure 2-12. All cores send requests as unicasts to a stateless directory home node (ordering point) which forwards it to all other cores via a broadcast. The requester collects all acknowledgements, and then unblocks the home node via a unicast. We enhance the protocol with an optimization that merges multiple read requests to the same cache line at the home node when those requests are competing for the same unblock message. This optimization reduces the additive queuing delay incurred by these waiting requests, and also avoids broadcasting each of them, lowering the application runtime of the baseline 3-

stage router network by 39.6% on average. Though HT commercially operates across multiple sockets (chips), we model and simulate a HT-based protocol running on a single multicore chip.

- **No Directory.** In these designs, there is no home node for data. The requester broadcasts all its requests, all other nodes snoop, and the owner (cache or memory) responds. A key requirement for these snoopy protocols is global ordering. On a bus-based design, where these protocols are highly prevalent, the central bus arbiter serves as the ordering point. On a distributed mesh, on the other hand, other techniques are required to guarantee race-free correct functionality in case of competing requests. Token Coherence [61] and INSO [8] are two techniques to run snoopy protocols on a mesh.

  We run **Token Coherence (TC)** for this thesis. Here each cache line is associated with a finite number of tokens. Readers hold at least one token, while writers need to hold all tokens. In Figure 2-13, we plot the coherence transactions and network messages for Token Coherence. L2 #8 broadcasts a write miss, and the owner L2# 14 responds with data and all its tokens, and invalidates its own copy. Sharer L2#0 snoops this request, invalidates its copy, and sends its token to L2 #8. Once L2 #8 receives both the data and all tokens for the line, it assumes ownership. Unlike HyperTransport, all nodes do not need to send ACKs/tokens, only the actual sharers do. In case of a race condition where two write requests for the same line originate at almost the same time and end up in tokens getting distributed between each, a timeout occurs. At this point one of the core statically assumes higher priority and sends out a *persistent request* demanding all tokens and gets the ownership.

Communication Patterns

For an N-core chip, we classify the communication patterns of protocols as 1-to-1, 1-to-M, and M-to-1 where M refers to multiple sources or destinations ($1 < M <= N$). 1-to-1 communication occurs in unicast requests/responses exchanged between cores.

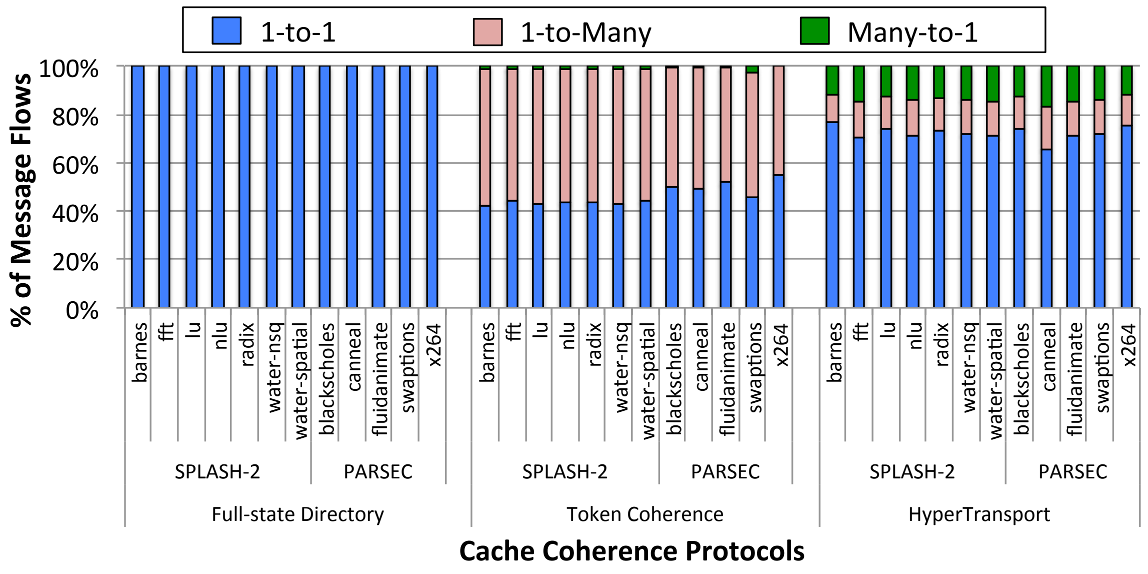


Figure 2-14: Message Flows in various cache coherence protocols.

1-to-M communication occurs in broadcasts and multicast requests [9, 1, 8, 75, 61]. M-to-1 communication occurs in acknowledgements [9, 1] or token collection [61, 71] in protocols to maintain ordering.

In this thesis, we run our network designs with a Full-state Directory, HyperTransport and Token Coherence, to study the interesting trade-off between storage requirements and number of network messages, and identify and correct network bottlenecks. Figure 2-14 shows a breakdown of the percentage of 1-to-1, 1-to-M and M-to-1 flows[6] in the network across SPLASH-2 [82] and PARSEC [14] benchmarks for these three protocols in a 64-core CMP. The evaluation setup will be discussed later in Section 2.3. For a Full-state Directory, 99% of messages are 1-to-1, with less than 1% 1-to-M corresponding to precise multicasts to invalidate sharers, with M = 19 on average. For HyperTransport, 1-to-M requests and M-to-1 responses form 14.3% and 14.1% of injected messages on average, respectively, with M = 64 in both cases. Token Coherence reduces M-to-1 traffic to 2%, with M = 12 on average, at the cost of a higher percentage (52.4%) of 1-to-M traffic (M = 64). These observations point to the criticality of the network fabric connecting the cores to efficiently handle *all three* kinds of communication, and not become a bottleneck.

[6]Every 1-to-M and M-to-1 flow translates to M messages in the network.

Virtual Networks (Vnets)

The series of messages sent by a coherence protocol as part of a coherence transaction fall within different **message classes**. For instance all directory protocols (full-state, partial-state and no-state) used in this thesis use 4 message classes: *request*, *forward*, *response* and *unblock*. Token Coherence, a snoopy protocol, uses 3 message classes: *request*, *response*, *persistent request*.

A potential deadlock can occur in the protocol if a request for a line from a L2 is unable to enter the network because the L2 it is waiting for a response for a previous request, while the response is unable to reach the L2 since all queues in the network are full of such waiting requests. To avoid such deadlocks, protocols require messages from different message classes to use different set of queues within the network. This is implemented by using **virtual networks (vnets)** within the physical network. Virtual Networks are identical to VCs in terms of their implementation: all vnets have separate buffers but multiplex over the same physical links. In fact many works on coherence protocols use the term virtual channels to refer to virtual networks. However, in this thesis we will strictly adhere to using the term virtual networks or vnets to refer to protocol level message classes. The number of vnets is thus fixed by the protocol. Each vnet, on the other hand, can have one or more VCs within each router, to avoid head-of-line blocking or avoid routing deadlocks, as discussed earlier in Section 2.1.3. In all NoC designs considered in this thesis, we will use the same number of VCs within each vnet.

Message sizes

We size our network parameters such that control messages (requests/forwards/unblocks) fit within one single flit, while data responses span multiple flits. For 128-bit flits which we assume in most of this thesis, unless specified, 64B cache line responses fit in 5 flits. Thus VCs within the request, forward or unblock vnets are 1-flit deep, while VCs within the response vnet are often more than 1-flit deep.

Point-to-Point Ordering

Certain message classes (and thus their vnets) require **point-to-point ordering** in the network for functional correctness. This means that two messages injected from the same source, for the same destination, should be delivered in the order of their issue. We implement point-to-point ordering for flits within ordered vnets by (i) using deterministic routing, and (ii) using FIFO/queuing arbiters for SA-i at each router. The first condition guarantees that two messages from the same source do not use alternate paths to the same destination as that could result in the older message getting delivered after the newer one if the former's path has more congestion. The second condition guarantees that flits at a router's input port leave in the order in which they came in.

In the protocols used in this thesis, only the persistent request vnet in Token Coherence requires point-to-point ordering. None of the other vnets in other protocols have this requirement as there are extra states and logic in the protocols to handle out-of-order messages.

2.2.2 Synthetic Traffic Patterns

In all our experiments, we assume all sources inject with a uniform random injection rate (without bursts), while the destination coordinates depend on the traffic pattern. Table 2.1 lists some common synthetic traffic patterns used for studying a mesh network, along with their average hop-counts and theoretical throughput with XY routing. The theoretical throughput or capacity is the injection rate at which some link(s) in the mesh is (are) sending 1-flit every cycle.[7] This is the best a topology can do, with perfect routing, flow control and microarchitecture.

[7]Table 2.1 shows that uniform random traffic offers the highest throughput, since it saturates when the bisection links of the mesh are fully occupied. For traffic patterns that saturate other links, throughput is lower.

Table 2.1: Synthetic Traffic Patterns for $k \times k$ Mesh

Source (binary coordinates): $(y_{k-1}, y_{k-2}, \ldots, y_1, y_0,\ x_{k-1}, x_{k-2}, \ldots, x_1, x_0)$

Traffic Pattern	**Destination** (binary coordinates)	**Avg Hops** (for $k = 8$)	**Throughput** (for $k = 8$) (flits/nodes/cycle)
Bit-Complement	$(\bar{y}_{k-1}, \bar{y}_{k-2}, \ldots, \bar{y}_1, \bar{y}_0, \bar{x}_{k-1}, \bar{x}_{k-2}, \ldots, \bar{x}_1, \bar{x}_0)$	8	0.25
Bit-Reverse	$(x_0, x_1, \ldots, x_{k-2}, x_{k-1}, y_0, y_1, \ldots, y_{k-2}, y_{k-1})$	5.25	0.14
Shuffle	$(y_{k-2}, y_{k-3}, \ldots, y_0, x_{k-1}, x_{k-2}, x_{k-3}, \ldots, x_0, y_{k-1})$	4	0.25
Tornado	$(y_{k-1}, y_{k-2}, \ldots, y_1, y_0, x_{k-1+\lceil \frac{k}{2} \rceil -1}, \ldots, x_{\lceil \frac{k}{2} \rceil -1})$	3.75	0.33
Transpose	$(x_{k-1}, x_{k-2}, \ldots, x_1, x_0, y_{k-1}, y_{k-2}, \ldots, y_1, y_0)$	5.25	0.14
Uniform Random	$random()$	5.25	0.5

2.3 Evaluation Methodology

We use the GEMS [62] + Garnet [7] infrastructure for all our evaluations, which provides a cycle-accurate timing model. Full-system simulations use Wind River Simics [4]. Network energy for each component is calculated using Orion 2.0 [43] and DSENT [76].

We model 64 in-order SPARC cores[8] in a tiled CMP with private 32 kB I&D L1 per tile and private/shared 1MB L2 per tile. We use the existing full-state directory, HyperTransport [9][9] and Token Coherence [61] implementations in GEMS. We use a 8×8 mesh network with 128-bit flits, and 1mm links. In all our NoC designs, unless specified, the number of buffers/VCs in each vnet is set by the buffer turnaround time.

We evaluate the parallel sections of the SPLASH-2 [82] and PARSEC [14] benchmarks for all configurations. Each run consists of 64 threads of the application running on our CMP. We run multiple times with small random perturbations to capture variability in parallel workloads [10], and average the results.

[8]Our CMP is homogeneous. However all the NoC architectures presented in this thesis can work for both homogeneous and heterogeneous cores since the core only acts as a traffic injector and receiver.

[9]HyperTransport was ported to GEMS from gem5 [16] for this thesis. Thanks to Brad Beckmann from AMD Research for implementing and optimizing the protocol's gem5 version.

2.3.1 Baseline and Ideal Networks

We will use the state-of-the-art **BASELINE (t_r=1)** NoC presented in Section 2.1.4 as the baseline design throughout this thesis. Chapter 3 will present the detailed pipeline and microarchitecture of this design. Chapters 4 and 5 will use alternate baselines with multicast support within routers.

We also use two ideal networks as yardsticks representing the best possible NoC design for the given Mesh topology.

- **IDEAL (t_r=1)** NoC is an idealized implementation of the BASELINE (t_r=1), with 1-cycle routers at every hop, and no contention along the route. This ideal is implemented by sending the flit with a fixed delay (calculated using the Manhattan distance between the source and destination nodes) from the source NIC to the destination NIC via an imaginary dedicated wire, instead of via the actual NoC.

- **IDEAL (T_N=1)** NoC models a NoC with 1-cycle delay for each message, irrespective of the hops traversed. This design is similarly modeled via imaginary dedicated wires that enable contention-less traversal.

The goal of the thesis is to design a real NoC with performance and power metrics close to those of the ideal NoCs.

2.3.2 Performance Metrics

We characterize the performance of NoC designs on three metrics.

- **Network Latency.** The target metric that this thesis is aimed at is network latency. As shown earlier in Equation 2.1, the network latency has a fixed component (router + link delay), a variable component (contention delay) and a serialization component. The thesis presents microarchitectural optimizations to reduce the fixed component from 1-cycle at every hop to 1-cycle throughout the network. The thesis also presents flow control optimizations to reduce congestion, which is usually dominant in collective communication (1-to-Many and Many-to-1) flows. Through synthetic traffic and full-system simulations, the thesis will demonstrate a low-load network latency of $\sim$1-cycle for flits, and a graceful degradation till saturation.

- **Network Throughput.** We define the **saturation throughput** of the network as the injection rate at which the network latency becomes $3\times$ the low-load latency. Throughput is a function of link utilization. Inefficient arbitration, buffer management and routing can lead to links going idle while there is waiting traffic and/or some links getting over-provisioned, leading to throughput loss. While the primary goal of this thesis is latency, most flow control techniques presented also try to push the saturation throughput closer to the theoretical capacity defined in Table 2.1 for synthetic traffic. For full-system traffic across all protocols and designs, we observe pretty low injection rates so the NoC is not throughput constrained as much as it is latency constrained. This is in part because our cores are in-order and non-speculative, and in part because most applications in SPLASH-2 and PARSEC have well behaved working sets that do not stress the cache subsystem a lot.

- **Full-system Runtime.** The full-system runtime is the runtime of the parallel section of our benchmarks, and our most important performance metric. A faster network by itself may not provide any returns if the messages whose delivery was speeded up are not on the critical path of the computation. In fact in some cases we observe faster NoCs result in higher cache miss rates since remote lines get invalidated faster before they can be used by their local cores. But by the same argument, in some cases a minor speedup in the network can provide enormous speedups at the full-system level if certain threads were able to get access to locks faster, or certain requests hit in remote caches before that line was evicted off-chip, and so on. The thesis thus uses full-system simulations instead of trace-driven ones. However, limitations in the GEMS [62] simulator restrict us from running simulations for greater than 64-cores.

2.4 Chapter Summary

This chapter provided necessary background on Networks-on-Chip for the reader to understand the various techniques presented in this thesis. The chapter also presented the different cache coherence protocols (in full-system runs) and synthetic traffic patterns we run atop our NoC. Evaluation methodology and target performance metrics were also discussed.

In the next chapter, we present the design of a state-of-the-art 1-cycle router which was silicon-proven in 90nm CMOS. This design will serve as the baseline for the rest of the thesis.

3 Single-cycle Per-hop NoC for 1-to-1 Traffic

Nobody gets to live life backward. Look ahead, that is where your future lies.
- Ann Landers

This chapter describes the detailed microarchitecture of a 1-cycle router (i.e., techniques to achieve t_r=1 in Equation 1.1). The design has been silicon proven in 90nm[1].

3.1 Introduction

Fabricated NoC prototypes in the past, such as UT Austin's 40-core TRIPS [31] and Intel's 80-core TeraFLOPS [37] have typically used relatively simple VC router architectures that do not optimize buffer utilization. Incoming data is always buffered without regard to route availability. This dissipates unnecessary energy, and introduces additional latency within a router pipeline. Even if the data-path is available, all packets go through multiple pipeline stages within each router, including buffering, routing, switch allocation, and switch (crossbar) traversal; this is then followed by the link (interconnect) traversal. We saw the opportunity to optimize the router control-path to utilize the available link bandwidth more efficiently by exploiting adaptive routing and intelligent flow control, thus enhancing throughput. This architectural enhancement allows many flits to bypass buffering, even at high traffic rates. By ob-

[1] A more optimized design in 45nm will be presented later in Chapter 4 in Section 4.6.

viating the need for buffer reads/writes, fewer buffers are required to sustain a target bandwidth. The result is a reduction in both the router latency and buffer read/write power. This approach has been explored in academic literature [65, 63, 57, 56], but has not made it to mainstream design.

In Chapter 2 Section 2.1.4 we described the microarchitecture of a NoC router, and the actions on the control and data-path. The control-path prepares the flit to traverse to the next router one-hop away. The actions performed are Route Compute (RC)[2], VC Allocation (VA) and Switch Allocation (SA). The data-path sends the actual flit (header and payload) through the crossbar switch to the next router's input buffers. The actions performed are Buffer Write (BW), Buffer Read (BR), Switch Traversal (ST) and Link Traversal (LT).

A decade of research in NoC pipelines has enabled all actions within the control-path to occur in parallel within one cycle[3]. However, the control-path and data-path for a particular flit are serialized, leading to 3-cycle routers (BW, SA//VA//RC, BR//ST) per-hop, followed by a 1-cycle LT. In this work, we design a router microarchitecture that parallelizes the control and data-paths. The basic idea is to send the route information of a flit in advance to the next router in control signals called *lookaheads*, one-cycle before the actual flit. Lookaheads setup the control-path for the incoming flit, allowing it to bypass BW and BR and proceed directly to ST and LT. Both the lookaheads and the flit spend a single-cycle within the router, and a single-cycle in the link between routers.

This work also addresses the increasing problem of NoC power consumption. In Intel's recent 80-core TeraFLOPS router, 39% of the 2.3W power budget of a tile was found to be consumed by the network. The primary contributers to NoC power are the buffers (31% in MIT RAW [77], 35% in UT TRIPS [31], 22% in Intel TeraFLOPS [37]), which are typically built out of SRAM or register files; the crossbar switches (30% in MIT RAW [77], 33% in UT TRIPS [31], 15% in Intel TeraFLOPS [37]); and the core-core links (39% in MIT RAW [77], 31% in UT TRIPS [31], 17% in Intel

[2]RC is performed one-hop in advance [27]

[3]In case of contention, SA and VA can fail leading to multiple cycles in the control-path.

TeraFLOPS [37]). Buffers are necessary to facilitate sharing of links, the crossbar is necessary to allow incoming flows to make arbitrary turns cycle-by-cycle, and the tile-to-tile links are necessary to deliver the payload to its destination. In this work, we leverage Token Flow Control (TFC) [56]-based adaptive routing and bypass flow control to allow flits to bypass buffer writes/reads even at high loads, reducing the buffer power. We also embed custom low-swing circuits within the crossbar and links to reduce data transmission power.

We fabricate a prototype NoC called the SWing-reduced Interconnect For a Token-based Network-on-Chip (SWIFT NoC) [53, 69]. The router microarchitecture of SWIFT is designed in RTL and implemented using a conventional 90nm, 1.2V CMOS standard cell library provided by the foundry. The synthesized router is manually integrated with the custom crossbar and links. The targeted NoC is an 8×8 2-D mesh, while the fabricated test chip is comprised of a 2×2 mesh subsection of four routers that verifies the design practicality and validates the simulated results.

This work makes the following contributions to the field of NoC design:

- This is the first fabricated NoC that incorporates unconventional microarchitectural optimizations, such as buffer-less bypassing and adaptive routing, and realizes them in a single-cycle router pipeline. This implementation results in 39% lower latency and 49% lower buffer power while sustaining the same throughput as a baseline[4] with uniform random traffic.

- This is the first fabricated NoC that uses reduced-swing interconnects in both the crossbar and the links. Data path transmission power is lowered by 62%, when compared with a baseline[4] design.

- Our detailed discussions on the design choices we made and corresponding impact on critical path delay, area, power and overall network performance may present a useful case study for NoC researchers and designers.

The total power savings of the entire network was measured to be 38%.

[4]The baseline router used for all comparisons was modeled similar to [37, 31] and is described in Section 3.2.

The rest of the chapter is organized as follows. Section 3.2 describes the baseline non-bypass pipeline, and Section 3.3 describes the SWIFT router microarchitecture. Section 3.4 details the reduced-swing crossbar and link designs. Section 3.5 highlights the features added to the chip that facilitate testing and measurements. Section 3.6 presents and analyzes our measured results. Section 3.7 discussed related work, and Section 3.8 concludes.

3.2 Baseline Non-Bypass Pipeline

The baseline router pipeline is 3 stages long (i.e., $t_r = 3$), followed by 1 cycle in the link (i.e., $t_w = 1$). The various stages are described next, along with the blocks that are active during that stage.

Stage 1-Buffer Write (BW). The incoming flit is written into buffers at each input port, which were implemented with register files generated from the foundry memory generators. These input buffers are organized as a shared pool among multiple VCs. The addresses of the buffers are connected as a linked list. An incoming flit that requires a free buffer obtains the address from the head of the linked list, and every buffer that is freed up appends its address to the tail of the linked list. One buffer is reserved per VC in order to avoid deadlock. Compared to private buffers per VC which can be implemented as a FIFO, our shared buffer design incurs an overhead of storing the read addresses of all flits in the VC state table, but has the advantage of reducing the numbers of buffers required at each port to satisfy buffer turnaround time[5].

Stage 1-Switch Allocation-Inport (SA-I). An input VC is selected from each input port to place a request for the switch. This is implemented using V:1 round robin arbiters at each input port, where V is the number of VCs per port. Round robin arbiters are simple to implement [22] and ensure that every VC gets a chance to send a flit.

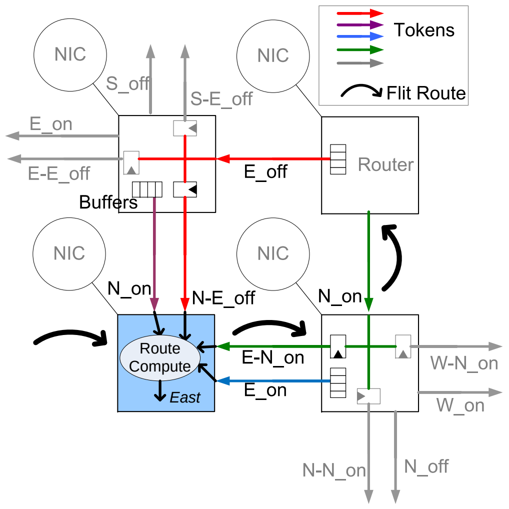
[5]Minimum number of cycles before which same buffer can be reused, as described earlier in Section 2.1.3

Stage 2-Switch Allocation-Outport (SA-O). The winners of SA-I at each input port place requests for their corresponding output ports. As no u-turns are allowed, there can be a maximum of 4 input ports requesting the same output port. These conflicts are resolved using 4:1 matrix arbiters, one for each output port. Matrix arbiters are used for fair allocation of the crossbar output port to all input ports [22]. Separating switch allocation into two phases of simpler arbitration, SA-I and SA-O, is a common approach to satisfy minimum cycle time constraints [67]. Note that a flit may spend multiple cycles in switch allocation due to contention.

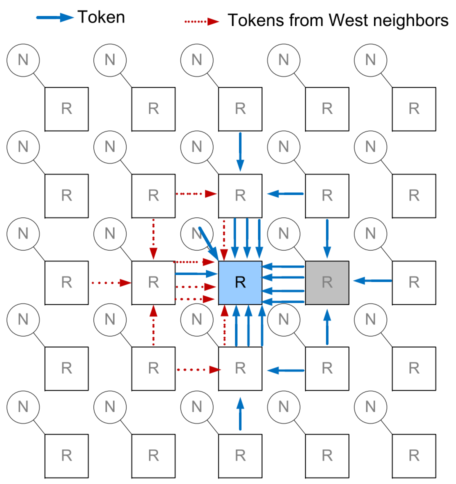
Stage 2-VC Allocation (VA). At the end of SA-O, winning head flits are assigned an input VC for their next hop. (Body and Tail flits follow on the same VC). VC allocation in our design is a simple VC selection scheme [55]. Each output port maintains a queue of free VCs at the input port of the next router. A switch request is allowed to be placed for an output port only if the router connected to that output port has at least one free input VC. The winning head flit of a switch output port, at the end of SA-O, picks up the free VC at the head of the queue and leaves. Thus there is no arbitration required, simplifying the VC allocation process. If the router receives a signal indicating a free VC from the next router, the corresponding VC id is enqueued at the tail of the queue. VA does not add any extra delay to the critical path since the updating of the queue and the computation of the next free VC id take place in parallel to SA-O.

Stage 2-Buffer Read (BR). Flits that won SA-I start a pre-emptive read of the buffers, in parallel to SA-O. This is because the register files require all input signals to be ready before the clock edge. If we wait until SA-O declares the winner of the switch output port, BR would have to be pushed to the next cycle, adding latency. The drawback of this is that there are wasted reads from the buffer which would consume power. We solve this by biasing SA-I to declare the same input VC as the winner until it succeeds to use the crossbar. This ensures that the same address is read out of BR to avoid additional switching power.

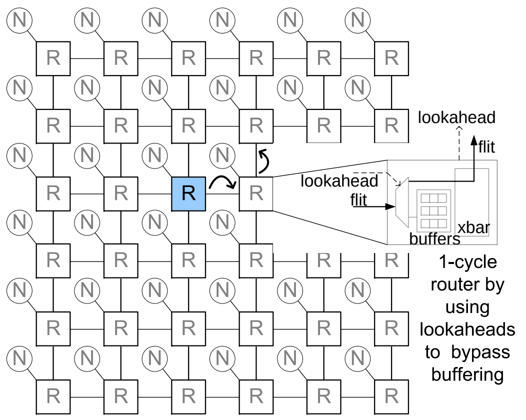
Stage 3-Switch Traversal (ST). The flits that won the switch ports traverse the crossbar switch.



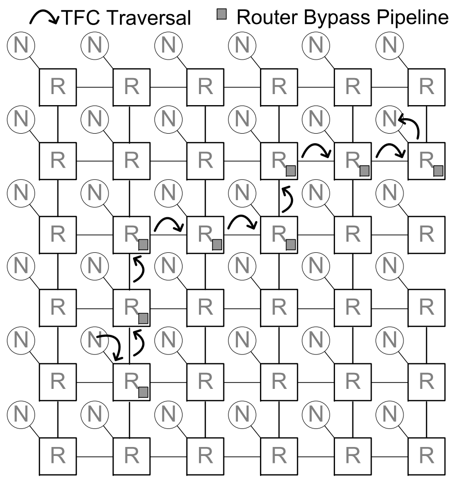
(a) **Routing using Tokens.**



(b) **Token Distribution.**



(c) **Bypass flow control using lookaheads.**



(d) **Traversal Example.**

Figure 3-1: Overview of the SWIFT NoC.

Stage 4-Link Traversal (LT). The flits coming out of the crossbar traverse the links to the next routers.

3.3 SWIFT Router Microarchitecture

Here we describe the architectural design of SWIFT NoC in detail, which is based on token flow control (TFC) [56], providing relevant background where necessary. The SWIFT router separates the control-path managed by *lookaheads* from the flit's data-path. The router implements 2 pipelines, a *non-bypass pipeline* which is 3-stages long and the same as the state-of-the-art 3-cycle baseline described in Section 3.2, as well as a *bypass pipeline*, which is one-cycle within the router and consists of just the crossbar traversal for flits (data) and switch allocation for lookaheads (control). This is followed by the link traversal.

We start by describing two primary control logic components that facilitate this bypass action, namely *tokens* and *lookaheads*.

3.3.1 Routing with Tokens

In the SWIFT NoC, every input port sends a one-bit token to its neighbor, which is a hint about buffer availability at that port. If the number of free buffers is greater than a threshold (which is three in order to account for flits already in flight), the token is turned ON (by making the wire high), else it is turned OFF. The neighbor broadcasts this token further to its neighbors, along with its own tokens. Flits use these tokens to determine their routes. They try to adapt their routes based on token availability. Figure 3-1(a) shows an example of this. The shaded router receives tokens from its N, E, and NE neighbors. The incoming flit chooses the East output port over the North output port based on token availability. We implement minimal routing, with a west-first turn rule, which was described in Chapter 2 Section 2.1.2, to avoid deadlocks. Any other adaptive routing algorithm can be used as well.

Each token is forwarded up to three-hops, via registers at each intermediate router. Tokens are also forwarded up to the network interfaces (NIC) at each router. The number three was fixed empirically based on simulations which can be found in the TFC paper [56]. Intuitively, deeper token neighborhoods do not help much since the information becomes stale with each hop. Moreover, the route is updated at every hop based on the tokens at that router, and the flit only needs to choose between a maximum of two output ports (for minimal routing). Adding more tokens would add more wires and registers without returning much benefit.

For illustration purposes, Figure 3-1(b) shows the token distribution relative to the shaded router in a two-hop neighborhood. 16 tokens enter the shaded router from a two-hop neighborhood, plus one from the local port. However, West-first routing algorithm allows us to remove tokens from the west neighborhood (except the immediate neighbor) since a packet has to go west irrespective of token availability, reducing the total tokens from (16 + 1) to (11 + 1). Similarly, there are a total of (36 + 1) tokens in a three-hop neighborhood. Removing the west tokens allows us to reduce this number to (22 + 1) bits of tokens per router and these act as inputs to the combinational block that performs route computation.

3.3.2 Flow Control with Lookaheads

Conventional flow control mechanisms involve arbitration for the crossbar switch among the buffered flits. Some prior works [65, 63, 57, 56] propose techniques to allow flits that have not yet arrived to try and pre-allocate the crossbar. This enables them to bypass the buffering stage and proceed directly to the switch upon arrival. This not only lowers traversal latency, but also reduces buffer read/write power. The SWIFT NoC implements such an approach, based on TFC [56], as shown in Figure 3-1(c). TFC has been shown to be better than other approaches like express virtual channels (EVC) [57] as it allows flits to chain together tokens to form arbitrarily long bypass paths with turns, while EVC only allowed bypassing within a dimension up to a maximum of three-hops. Other approaches to tackle buffer power include adding physical links to bypass intermediate routers [48], or using link repeaters as temporary buffers [64] to reduce buffers within the router. These techniques can enhance energy-delay further at the cost of more involved circuit design.

In the SWIFT NoC, the crossbar is pre-allocated with the help of lookahead signals, which are 14-bit signals sent for each flit, one-cycle before it reaches a router. The implementation of the lookahead generation and traversal to enable a one-cycle advanced arrival will be explained later in Section 3.3.3.

A lookahead is prioritized over locally-buffered flits, such that a local switch allocation is killed if it conflicts with a lookahead. If two or more lookaheads from different input ports arrive and demand the same output port, a switch priority pointer at the output port (which statically prioritizes each input port for an epoch of 20 cycles for fairness) is used to decide the winner and the other lookaheads are killed. The flits corresponding to the killed lookaheads get buffered similar to the conventional case. Since the bypass is not guaranteed, a flit can proceed only if the token from the neighbor is ON (indicating an available buffer).

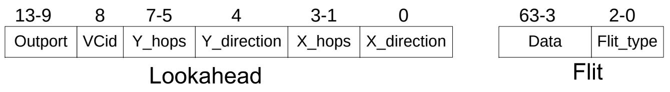


Figure 3-2: Lookahead and Flit Payloads.

The lookahead and flit payloads are shown in Figure 3-2. Lookaheads carry information that would normally be carried by the header fields of each flit: destination coordinates, input VC id, and the output port the corresponding flit wants to go out from. They are thus not strictly an overhead. Lookaheads perform both switch allocation, and route computation.

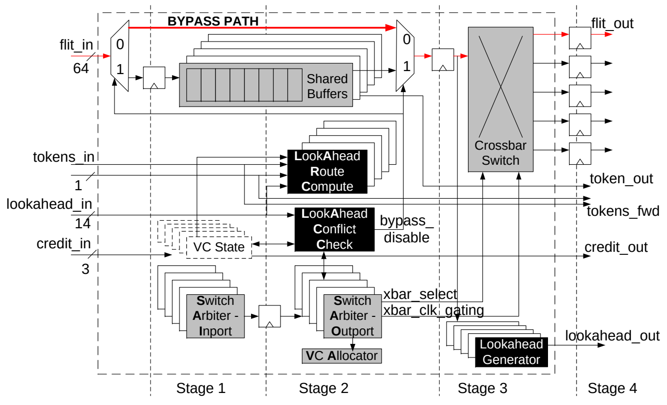
The SWIFT flow control has three major advantages over previous prototypes with simple flow control.

- **Lower Latency**: Bypassing obviates the buffer write, read, and arbitration cycles.

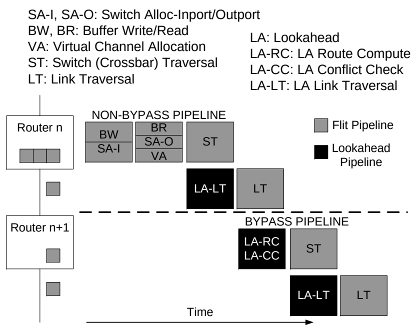
- **Fewer buffers**: The ability of flits to bypass at all loads keeps the links better utilized while minimizing buffer usage, and reducing buffer turnaround times. Thus, the same throughput can be realized with fewer buffers.

- **Lower power**: Requiring fewer buffers leads to savings in buffer power (dynamic and leakage) and area, while bypassing further saves dynamic switching energy due to a reduction in the number of buffer writes and reads.

The SWIFT NoC guarantees that flits within a packet do not get re-ordered. This is ensured by killing an incoming lookahead for a flit at an input port if another flit from the same packet is already buffered. Point-to-Point ordering is however not guaranteed by SWIFT. This is because lookaheads are prioritized over locally buffered flits, which could result in two flits from the same source to the same destination getting re-ordered if the first one happened to get buffered at some router while the second



(a) **SWIFT Router.**



(b) **SWIFT pipelines.**

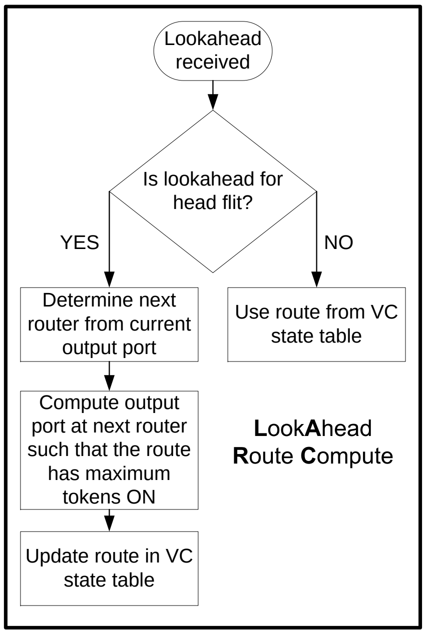
Figure 3-3: SWIFT Router Microarchitecture and Pipelines.

one succeeded in bypassing that router. Most NoC designs use multiple virtual networks to avoid protocol level deadlocks. While request virtual networks often require point-to-point ordering for consistency reasons, response virtual networks often do not place this constraint, and TFC can be used within these virtual networks. A potential network traversal in the SWIFT NoC using tokens and lookaheads is shown in Figure 3-1(d).

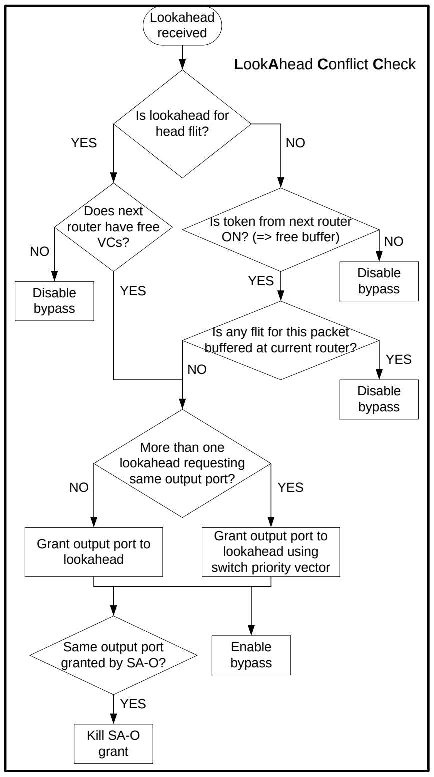
3.3.3 Router Microarchitecture

SWIFT tries to present a one-cycle router to the data by performing critical control computations off the critical path. The modifications over a baseline router are highlighted in black in Figure 3-3(a). In particular, each SWIFT router consists of two pipelines: a non-bypass pipeline which is three-stages long (and the same as a state-of-the-art baseline described in Section 3.2), and a bypass pipeline, which is only one-stage and consists of the crossbar traversal along the data (flit) path, and switch allocation along the control (lookahead) path. The router pipeline is followed by a one-cycle link traversal.

Figure 3-3(b) shows the pipeline followed by the lookaheads to enable them to arrive a cycle before the flit, and participate in the switch allocation at the next router. All flits try to use the bypass pipeline at all routers. The fall-back is the baseline three-stage non-bypass pipeline.


Lookahead received
Is lookahead for head flit?
YES
NO
Determine next router from current output port
Use route from VC state table
Compute output port at next router such that the route has maximum tokens ON
LookAhead Route Compute
Update route in VC state table

(a) **LA-RC.**


Lookahead received
LookAhead Conflict Check
Is lookahead for head flit?
YES
NO
Does next router have free VCs?
NO
Disable bypass
YES
Is token from next router ON? (=> free buffer)
NO
Disable bypass
YES
Is any flit for this packet buffered at current router?
YES
Disable bypass
NO
More than one lookahead requesting same output port?
NO
YES
Grant output port to lookahead
Grant output port to lookahead using switch priority vector
Same output port granted by SA-O?
Enable bypass
YES
Kill SA-O grant

(b) **LA-CC.**

Figure 3-4: Flow Chart of Lookahead Pipeline Actions.

Lookahead Route Compute (LA-RC.) The lookahead of each head flit performs a route compute (LA-RC) to determine the output port at the *next* router [27]. This is an important component of bypassing because it ensures that all incoming flits at a router already know which output port to request, and whether to potentially proceed straight to ST. We used West-first routing, an adaptive-routing algorithm that is deadlock free [22]. The adaptive-routing unit is a combinational logic block that computes the output port based on the availability of the tokens from 3-hop neighboring routers, which is explained later, rather than use local congestion metrics as indication of traffic. An overview of LA-RC is shown in Figure 3-4(a).

Lookahead Conflict Check (LA-CC). The lookahead places a request for the output port in the LA-CC stage, which grants it the output port unless there is a conflict or the output port does not have free VCs/buffers. An overview of LA-CC is shown in Figure 3-4(b). LA-CC occurs in parallel to the SA-O stage of the non-bypass pipeline, as shown in Figure 3-3(a). A lookahead is given preference over the winners of SA-O, and conflicts between multiple lookaheads are resolved using the switch priority vector described earlier in Section 3.3.2. Muxes connect the input ports of the winning lookaheads directly to the crossbar ports. The corresponding flits that arrive in the next cycle bypass the buffers, as shown in Figure 3-3(a). Any flits corresponding to killed lookaheads, meanwhile, get buffered and use the non-bypass pipeline.

Lookahead Link Traversal (LA-LT). While the flit performs its crossbar traversal, its lookahead is generated and sent to the next router. All the fields required by the lookahead, shown in Figure 3-2, are ready by the end of the previous stage of LA-RC and LA-CC. Figure 3-3(b) shows how the lookahead control pipeline stages interact with the flit pipeline stages in order to realize a one-cycle critical data-path within the router.

3.4 Low-Voltage Swing On-Chip Wires

The custom reduced-swing interconnect circuits for the SWIFT NoC were designed by Prof Patrick Chiang and his PhD student Jacob Postman at Oregon State University. This section is included in this thesis for completeness.

SWIFT's bypass pipeline has two stages, switch and link traversal, corresponding to the data-paths through the crossbar switch and the core-to-core interconnect respectively. Both are critical NoC components and major contributors to power and area budgets, amounting to 32% of the network power and nearly 10% of total chip area [37].

Unlike locally connected logic cells that communicate through short, locally-routed wires, crossbar and link interconnect exhibit long distances and close inter-wire coupling, such that dynamic power consumption is dominated by large wire capacitances rather than gate input capacitances. To reduce the power required to drive these large capacitive wire loads, reduced-voltage swing signaling was implemented using dual voltage supply differential reduced-swing drivers (RSD) (Figure 3-5(a)), followed by a simple sense-amplifier receiver (Figure 3-5(d)). The lower supply voltage is gen-

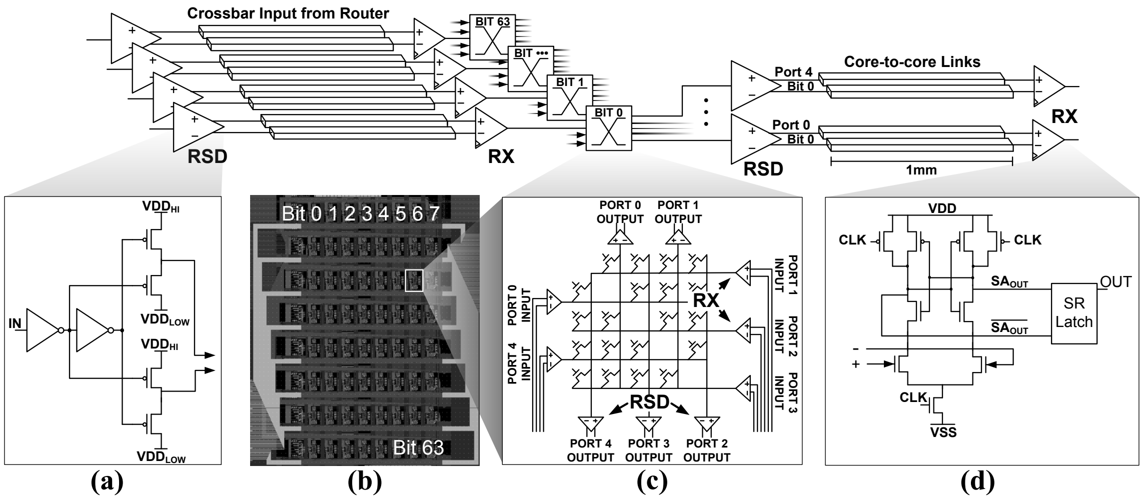


Figure 3-5: Crossbar and link circuit implementation (a) Reduced Swing Driver (RSD). (b) Bit-slice array crossbar layout. (c) Crossbar bit-slice schematic with link drivers at slice output ports. (d) Clocked sense amplifier receiver (RX).

erated off-chip, allowing for signal swings to be adjusted easily during testing as the difference between the two supplies. In practice, voltage supplies 0.2V to 0.4V below the core logic supply are often already available on-chip for SRAM caches or other components that operate at a reduced supply voltage. Occupying 7.8um^2 and 15.2um^2 respectively, the same driver and receiver are used in both the crossbar and link designs.

While differential signaling approximately doubles the wire capacitance of each bit by introducing a second wire, it removes the necessity of multiple buffer stages and enables the use of reduced voltage swings, resulting in quadratic power savings in the data-path. For example, if the power required to drive a wire is given by Equation (3.1), then the power required to drive the differential wires at 200mV is approximately given by Equation (3.2).

$$P_{(swing=1.2V)} = \frac{1}{2} C_{wire} V^2 f \tag{3.1}$$

$$P_{(swing=200mV)} = \frac{1}{2}(2C_{wire})\frac{1}{36}V^2 f = \frac{1}{18}P_{(swing=1.2V)} \tag{3.2}$$

Hence, reducing the voltage swing from 1.2V to 200mV results in greater than 94%

reduction in the power required to drive the interconnect wire. At 200mV swing, more than 98% of the crossbar and link power is consumed by the clock distribution, switch selection signals, driver input capacitance, and the sense amplifiers used to resolve the low-voltage swing signals. Further reductions in voltage swing require either larger transistors or offset correction in the receiver to overcome input offset, diminishing the return on additional voltage swing reduction in the data-path. The area-efficient sense amplifiers, shown in Figure 3-5(d), are designed with near-minimum sized transistors and with approximately 100mV simulated 3σ offset are used rather than larger or more complex devices that achieve better input sensitivity.

3.4.1 Reduced-Swing Link

A significant body of work exists exploring high-speed, energy-efficient, on-chip interconnects that attempt to achieve the lowest mW/Gbps on narrow interconnect wires across distances up to 10mm. However a typical 2D-mesh NoC is likely to require wide, parallel links spanning a distance of just 1-2mm [37], limiting the practicality of previously proposed on-chip interconnects. For instance, while Bae *et al.* [11] and Kim *et al.* [46] achieve 3Gbps and 4Gbps on a 10mm link, the transceivers occupy 887um^2 and 1760um^2 respectively. By contrast, in this design, the combined driver and receiver area is only 23um^2, allowing the many transceivers required for a wide data bus to be realized in an acceptable area footprint.

A major concern for the proposed interconnect is susceptibility to crosstalk interference from digital logic signals on lower metal layers, as reduced-swing signals are particularly sensitive to coupled noise from nearby full-swing wires. To address this, shielding wires were routed on M6 between and in parallel to the differential pairs on M7. This differential-mode shielding approach adds less capacitance to the signal wires than routing ground shielding under the entire link, while still minimizing crosstalk from signals that would couple asymmetrically on to one of the differential wires. Worst case simulated differential mode crosstalk from a 1mm full swing aggressor signal is reduced from 128mV without shielding to 29mV with shielding, providing sufficient voltage margin to preserve signal integrity at 200mV signal swing and 100mV receiver input offset.

3.4.2 Reduced-Swing Crossbar

The simplest and most obvious crossbar layout is to route a grid of vertical and horizontal wires with pass-gates or tri-state buffers at their intersection points, as shown in Figure 3-5(c). While simple, this approach suffers from a number of major disadvantages including poor transistor density, low bandwidth and a n^2 bit-to-area relationship. Higher crossbar speeds and improved density can be achieved using mux-based switches that place buffered muxes throughout the area of the crossbar or by implementing crossbar pipelining to improve speed by allowing sections of the wire load to be driven in separate clock cycles. While simple to implement in digital design flows, both approaches introduce additional loading in the form of increased fanout buffering and clock distribution that results in increased power consumption.

Based on these observations, the crossbar implemented in this design improves both performance and energy efficiency by replacing crossbar wires with low-swing interconnect. This approach seeks to drive as much of the large wire capacitances of the crossbar as possible with a reduced voltage swing, without introducing additional clocked elements or buffers. Implemented as a bit-sliced crossbar, each of the 64-bits in each of the five input buses is connected to a one-bit wide, 5-input to 5-output crossbar, along with the corresponding bits from each of the other four ports. An 8×8 grid is then patterned out of 64 of these bit-cell crossbars in order to construct a 64-bit wide, 5x5 crossbar as shown in Figure 3-5(b).

Each crossbar bit-slice consists of 5 sense amplifiers, 20 pass-gates and 5 RSDs as shown in Figure 3-5(c). Each of the five reduced-swing differential inputs is driven at the periphery of the crossbar by an RSD connected to the output of the router logic. At the positive clock edge, each of the five low-swing differential inputs is converted to full-swing logic by the sense amplifier and drives a short 6um wire through a pass-gate transistor, and then into the interconnect RSD at one of the four possible output ports (U-turns are not allowed). The receiver, which consists of a near-minimum sized, 9-transistor sense-amplifier followed by a minimum-sized NAND-SR

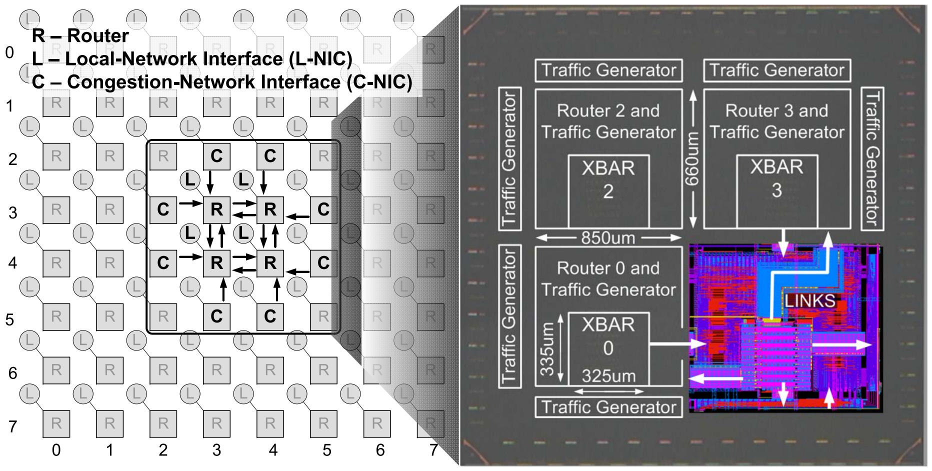

R – Router
L – Local-Network Interface (L-NIC)
C – Congestion-Network Interface (C-NIC)
Traffic Generator
Router 2 and Traffic Generator
XBAR 2
Router 3 and Traffic Generator
XBAR 3
660um
850um
Router 0 and Traffic Generator
XBAR 0
335um
325um
LINKS

Figure 3-6: 2x2 SWIFT NoC prototype overview and die photo, overlaid with node 1 layout. Specs: Technology=90nm, Freq=400MHz, Voltage=1.2V, Area=4mm^2, Transistors=688K

latch, acts as a DFF with low-swing differential inputs, replacing the flip-flop that would otherwise reside at the output of the crossbar-traversal pipeline stage. Like mux-based crossbars, this crossbar topology results in irregular data-paths across the 64b parallel interconnect, requiring that the maximum crossbar speed be bounded by the longest data-path delay through the crossbar. Full-swing select signals are routed on M1 and M2, differential data signals are routed on minimum width wires on M3-M5, and a separate clock wire is routed on M7 for each port. The clock is custom-routed to closely match the worst case RC delay of the data-path in order to minimize clock skew.

3.5 Design features for Testability

In this section, we describe the various features added to the design to facilitate testing and measurement.

3.5.1 Network Interfaces (NIC)

Each router is connected to a Local Network Interface (L-NIC) which houses a *traffic injector* and a *traffic receiver*.

The *traffic injectors* generate uniform random traffic (traffic to destinations that are randomly generated using a PRBS generator), at an injection rate specified via a scan chain. They also generate lookaheads based on tokens. To avoid the need for buffering, the traffic injectors generate packets one at a time[6]. Each traffic injector uses a 32-bit counter that signifies the current time stamp, synchronized via the global reset signal. Data flits carry their generation time in the data field to aid in the calculation of flit latency.

The *traffic receivers* at the L-NICs receive flits and send back the free VC and On/Off token signals. In addition, they compute the total received packets and total packet latency.

For the 2×2 mesh test chip, in addition to the L-NICs, each router includes Congestion Network Interfaces (C-NICs) connected to its two *unconnected* ports at the edges of the mesh. These are added to enable simulations of the router with varying activity at *all* ports. The C-NICs also incorporate a traffic injector and a traffic receiver. They also send out tokens to emulate a larger mesh. Overall we have 12 traffic injectors on the chip, as shown in Figure 3-6.

In our test chip, the traffic injectors work in two modes: *non-congestion* and *congestion*. In the *non-congestion* mode, only the 4 traffic injectors in the L-NICs inject packets to be sent to the other three L-NICs in the 2×2 network. In the *congestion* mode, we try to mimic the traffic within the 2×2 network as if it were at the center of an 8×8 mesh. The 4 L-NICs and the 8 C-NICs inject traffic meant for the other 63 nodes in the hypothetical 8×8 network. Injection rate at the C-NICs is set at double the L-NIC injection rate to correctly model the channel load of one slice of an 8×8 mesh [22].

[6]The traffic generator does not have extra buffers to store packets that are generated but cannot go into the network due to congestion. Thus it only supports one packet at a time, which means that packet generation cannot take place if there is another packet waiting to be injected. The artifact of this is that queuing delay at the network interface is not modeled and thus at high input injection rates, packet latencies saturate instead of rising to very high values.

Table 3.1: Modes of operation

Parameter	Description
Test Network	Send only one test packet to specified destination
Bypass Enable	Enable/Disable bypassing in the routers
Congestion Traffic	Send congestion traffic from L-NICs and C-NICs
Clk Gating Enable	Allow clock gating in crossbar
Stats Enable	Enable statistics collection at the NICs

Table 3.2: Outputs from chip

Register	Bits	Description
Simulation Time	32	Total cycles for which the simulation was run
Total latency	32	Total received packet latency
Total packets	32	Total received packets
ni_destination_error	1	Flit received at wrong destination L-NIC
ni_ooo_error	1	Flits of packet received out of order at L-NIC
rtr_route_error	5	Flits of same packet requesting different output ports at router inport
rtr_ooo_error	5	Flits of same packet arrived out of order at router inport

3.5.2 Error Detection

Specific error bits are set at each of the four routers and the four L-NICs if flits are received out-of-order or at incorrect destinations. All of these error bits are scanned out at the end of the simulation to determine the frequency at which the chip fails.

3.5.3 Scan Chains

A 1640-bit scan chain connects through the 12 NICs in the 2×2 chip to set the various parameters (PRBS seeds, the injection rates, and the modes of operation) at the beginning of a simulation. *Bypassing*, *clock-gating* and *congestion* modes can each be enabled or disabled. The various modes of operation of the network are specified in Table 3.1. At the end of the simulation, the scan chain reads out the packet statistics (total packets received, total latency, simulation cycles) and the error bits, as shown in Table 3.2.

Table 3.3: Comparison of NoC designs

Parameter	TilePro64 [81]	TeraFLOPS [37]	TRIPS [31]	Baseline*	SWIFT
Process parameters					
Technology	90nm	65nm	130nm	90nm	90nm
Frequency	700-866 MHz	5GHz	366 MHz	NA	400 MHz
Router Area	Not available	$0.34mm^2$	$1.10mm^2$	$0.48^\dagger mm^2$	$0.48mm^2$
Network parameters					
Topology	8×8 mesh	8×10 mesh	4×10 mesh	8×8 mesh	8×8 mesh‡
Flit size	32b	39b	138b	64b	64b
Msg Length	1-128 flits	2+ flits	1-5 flits	5 flits	5 flits
Routing	XY	Source	YX	XY	West-first
Flow Ctrl	Wormhole	VC	VC	VC	TFC [56]
Buff Mgmt	Credit	On/Off	Credit	On/Off	TFC [56]
Router parameters					
Ports	5	5	6	5	5
VCs/port	0 (5 nets)	2	4	2 and 4	2
Buffers/port	12 (3/dyn net)	32	8	8 and 16	8
Crossbar	5x5	5x5	6x6	5x5	5x5

* Not fabricated, only laid out for comparison purposes.
† BASELINE ($t_r = 3$) tile was given same area as SWIFT for place-and-route.
‡ 2×2 mesh for test chip.

3.6 Evaluations

In this section, we report both the simulated and measured results of the SWIFT NoC prototype, and compare it to a BASELINE (t_r=3) NoC.

3.6.1 SWIFT NoC

SWIFT NoC parameters are shown in Table 3.3. We chose eight buffers per port, shared by 2 VCs. This is the minimum number of buffers required per port, with one buffer reserved per VC for deadlock avoidance and six being the buffer turnaround time with onoff signaling between neighboring routers. We used standard-cell libraries provided by ARM Corporation for synthesis. The place and route of the router RTL met timing closure at 600 MHz. The process technology used and use of standard cells instead of custom layouts, limits our router design from running at GHz speeds, such as in Intel's TeraFLOPS Router [37]. Note that based on extracted layout simulations, the custom reduced-swing transceivers are designed to operate at 2 GHz across 1-mm distances with 250-mV voltage swing. In an actual Chip-Multi Processor, a tile consists of both the processing core, and a router, with the router accounting for less than a quarter of the tile area [37]. Since we do not integrate processing cores in this design, we place the routers next to each other for area reasons. This results in asymmetric link lengths in our chip, but we size each driver and sense amplifier for $1mm$ links. A photo of our prototype test chip overlaid with the layout of node 1 is shown in Figure 3-6.

Due to the $4mm^2$ network size, we use a synchronous clock with tunable delays to each core rather than a globally-asynchronous, locally-synchronous approach as in the TeraFLOPS chip [37], which was outside the scope of this work. The test chip operates at 400MHz at low load, however is limited to 225MHz at high-injection due to voltage supply droop caused by excessive resistance in the power distribution.

3.6.2 Baseline NoC

To characterize the impact of the various features of the SWIFT NoC, we implemented a 3-stage baseline VC router similar to the one in UT TRIPS [31] and Intel TeraFLOPS [37] in the same 90-nm technology. It was described in Section 3.2. The non-bypass pipeline in SWIFT is the same as the baseline pipeline, thus allowing us to compare the performance, power, and area of the SWIFT and the baseline designs and the impact of our additions (bypass-logic and the reduced-swing crossbar).

Once we finalized the baseline router pipeline, we swept the number of VCs and buffers in the baseline such that the peak operating throughput of both the baseline and the SWIFT NoC was the same. This is described in Section 3.6.4. We use two configurations, BASELINE_2vc-8buf (t_r=3) and BASELINE_4vc-16buf (t_r=3), for network performance and power measurements.

3.6.3 Timing

Figure 3-7 shows the the critical paths of the SWIFT and baseline routers, and it occurs during the SA-O stage in both cases[7]. The baseline is 400ps faster. Disecting

[7]Our choice of the VC select scheme made the delay of the VC allocation stage trivial; if a separable VC allocator was used, like in TeraFLOPS [37], that would have been the critical path.

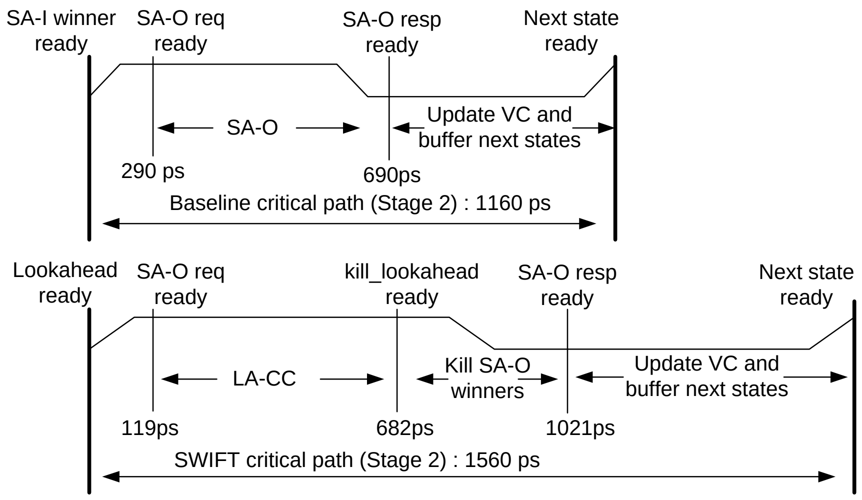


Figure 3-7: Critical Paths of the SWIFT router and BASELINE (t_r=3) router

the various components of the critical path gives interesting insights. The generation of the SA-O request signals, and the updating of the VC and buffer states is faster in SWIFT due to fewer number of VCs and buffers. The primary bottleneck in SWIFT turns out to be the 339ps incurred in killing the SA-O winners. The SWIFT router performs SA-O and LA-CC independently, in parallel, and then kills all the SA-O assignments which conflict with the lookahead assignments for the same input or output ports of the crossbar, to maintain higher priority for lookaheads. In hindsight, we could have done this faster by combining LA-CC and SA-O: muxing out requests from SA-I winners (buffered flits) if lookaheads arrived at those input ports, and then arbitrating for output ports using SA-O, biasing it to give preference to lookaheads. We adopt this optimization in a follow-on tapeout of a similar 1-cycle router in 45nm, and achieve a 1GHz frequency. This design will be presented later in Chapter 4 in Section 4.6.

3.6.4 Network Performance

Figure 3-8 demonstrates that the measured and simulated latency curves match, confirming the accuracy and functionality of the chip. The 2×2 network delivers a peak throughput of 113 bits/cycle.

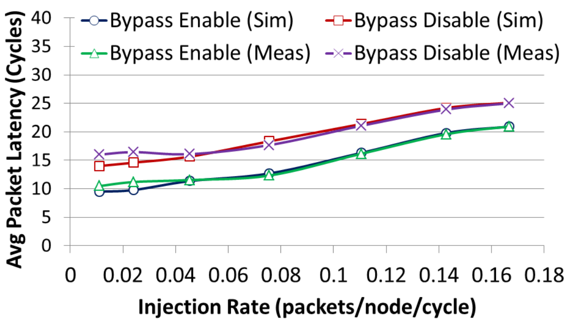
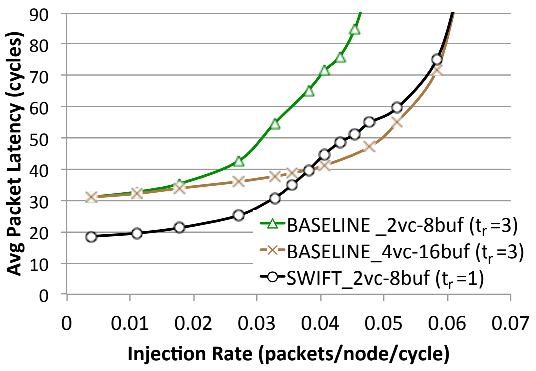
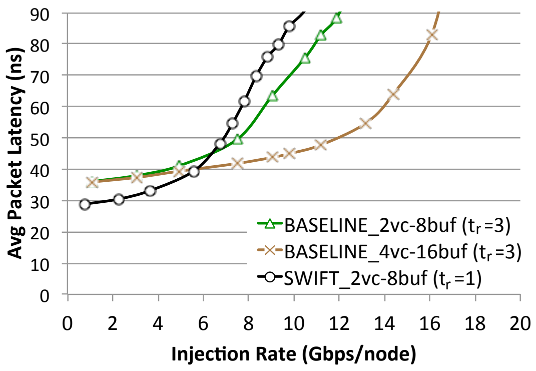


Figure 3-8: Network Performance Measurements from 2×2 Chip



(a) **Latency in cycles.**



(b) **Latency in ns.**

Figure 3-9: Network Performance of 8×8 SWIFT NoC.

Average Packet Latency (cycles). We first compare the average packet latencies of the 8×8 SWIFT NoC and the baseline NoC in cycles via RTL simulations. Figure 3-9(a) plots the average packet latency as a function of injection rate for SWIFT, and two interesting design points of the baseline: BASELINE_2vc-8buf (t_r=3) and BASELINE_4vc-16buf (t_r=3). At low loads, SWIFT provides a 39% latency reduction as compared to the baseline networks. This is due to the almost 100% successful bypasses at low traffic. At higher injection rates, BASELINE_2vc-8buf (t_r=3) saturates at 21% lower throughput. SWIFT leverages adaptive routing via tokens, and faster VC and buffer turnarounds due to bypassing, in order to improve link utilization which translates to higher throughput. BASELINE_4vc-16buf (t_r=3) matches SWIFT in peak saturation throughput (the point at which the average network latency is three times the no-load latency) in bits/cycle.

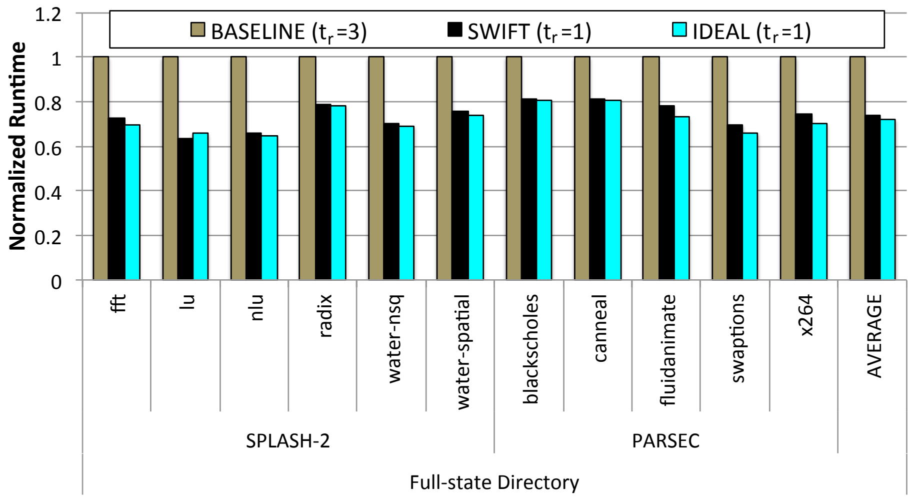


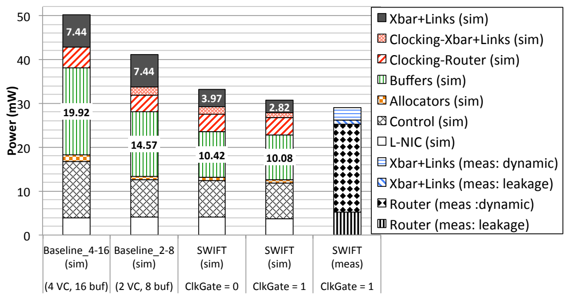
Figure 3-10: Full-System Runtime of SWIFT compared to BASELINE (t_r=3) and IDEAL (t_r=1) NoCs.

Average Packet Latency (ns). If we take the critical paths of SWIFT and the baseline in account, the latter network can run at a frequency 1.34 times faster than the former. Under this operating condition, Figure 3-9(b) shows the performance results of the 8×8 NoCs in nanoseconds, instead of cycles. The SWIFT NoC shows a 20% latency reduction at low-load as compared to the baselines, and similar saturation throughput as BASELINE_2vc-8buf (t_r=3).

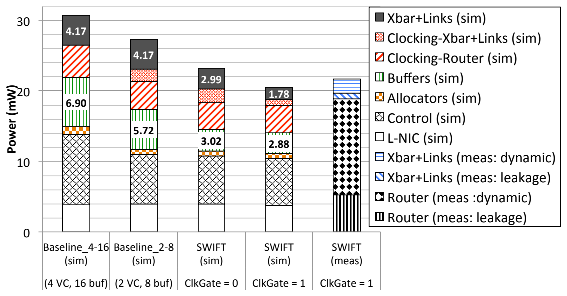
3.6.5 Full-system Performance

Figure 3-10 compares the full-system runtime of a CMP running a full-state directory protocol and implementing the SWIFT NoC against those implementing the BASELINE (t_r=3) and IDEAL (t_r=1) NoCs[8] across a suite of SPLASH-2 and PARSEC benchmarks. The configuration parameters of the simulation were described earlier in Section 2.3. The protocol uses 4 vnets (request, forward, response, unblock) and we give 4 VCs to each in all designs. SWIFT reduces runtime by 26% on average compared to the baseline, and is less than 2% away from the IDEAL ($t_r = 1$) NoC. The benchmarks do not stress the NoC much, and hence we did not observe any significant impact of fewer/more VCs, or of adaptive routing via tokens vs. XY routing.

[8]Section 2.3.1 describes the implementation of the contention-less ideal NoC.



(a) **High Traffic (1 packet/NIC/cycle).**



(b) **Low Traffic (0.03 packets/NIC/cycle).**

Figure 3-11: Measured Router Power at High and Low Injection.

3.6.6 Power

We compare the SWIFT and baseline routers at the same performance (throughput) points for fairness. In Section 3.6.4, we observed that BASELINE_4vc-16buf (t_r=3) matches SWIFT in saturation throughput if both networks operate at the same frequency. BASELINE_2vc-8buf (t_r=3) matches SWIFT in saturation throughput if it operates at a higher frequency, or if the networks are operating at low loads. We report power numbers for both BASELINE_2vc-8buf (t_r=3) and BASELINE_4vc-16buf (t_r=3) for completeness.

We perform power simulations and measurements at a frequency of 225 MHz and VDD of 1.2 V, and the results are shown in Figure 3-11(a) and 3-11(b) at high and low loads, respectively. In both graphs, all 12 traffic generator NICs are injecting traffic. The low-swing drivers were set to 300-mV signal swing. Because the L-NIC shares a supply with the router while the crossbar shares a supply with the reduced-swing links, it was not possible to measure each of the blocks separately. Instead, post-layout extracted simulations were performed to obtain an accurate relative breakdown of the power consumption of the different components, which were then compared and validated with chip measurements of the combined blocks.

At high loads, operating at the same frequency, BASELINE_4vc-16buf (t_r=3) matches SWIFT in performance, but has 49.4% higher buffer and 62.1% higher crossbar and link power. SWIFT (last two bars in Figure 3-11(a)) achieves a total power reduction of 38.7% at high injection, with the chip consuming a peak power of 116.5

mW. At low loads, operating at the same frequency, BASELINE_2vc8buf (t_r=3) can match SWIFT in performance, but consumes 24.6% higher power than SWIFT (last two bars in Figure 3-11(b)).

BASELINE_2vc-8buf (t_r=3) and SWIFT have the same VC and buffer resources. SWIFT adds buffer bypassing logic (using tokens and lookaheads), and the low-swing crossbar. Thus comparing BASELINE_2vc-8buf (t_r=3) and the first bar of SWIFT shows us that buffer bypassing reduces power by 28.5% at high loads, and 47.2% at low loads, while the low-swing data-path reduces power by 46.6% at high loads and 28.3% at low loads. These results are intuitive, as buffer write/read bypasses have a much higher impact at lower loads when their success rate is higher, while data-path traversals are higher when there is more traffic.

Lookahead signals allow the crossbar allocation to be determined a cycle prior to traversal, making per-port, cycle-to-cycle clock gating possible. Therefore, clock gating was implemented at each crossbars input port, using the crossbars forwarded clock, reducing the crossbar clock distribution power by 77% and 47%, and sense amplifier power by 73% and 43% at low and high injection, respectively.

The combined average energy-efficiency of the crossbar and link at the network saturation point is measured to be 128 fJ/bit, based on chip measurements of the crossbar and link currents, and the count of the received packets.

Overheads. The west-first adaptive routing logic used for tokens, the lookahead arbitration logic, and the bypass muxes account for less than 1% of the total power consumed in the router, and are therefore not a serious overhead. This is expected, as the allocators account for only 3% of the total power, consistent with previous NoC prototypes. The control power of the SWIFT NoC is observed to be 37.4% lower than BASELINE_4vc16buf (t_r=3), due to fewer buffers and VCs (hence smaller decoders and muxes) required to maintain the same throughput. The overall control power of SWIFT is approximately 26% of the entire router power, as seen in Figure 3-11. This high proportion is primarily due to a large number of flip-flops in the router, many of which were added conservatively to enable the design to meet timing constraints, and could have been avoided by using latches. In addition, the shared buffers require

Table 3.4: Area Comparison (Absolute and Percentage)

Component	SWIFT Area, % of total	BASELINE (t_r=3) Area, % of total
Tokens	1,235 um^2, 0.82%	0
Bypass	10,682 um^2, 7.10%	0
Buffers	72,118 um^2, 47.94%	81,231 um^2, 40.08%
Crossbar	15,800 um^2, 10.50%	21,584 um^2, 10.64%
Control	50,596 um^2, 33.63%	99,856 um^2, 49.27%
Total	150,431 um^2, 100%	202,671 um^2, 100%

significant state information in order to track the free buffer slots and addresses needed for each flit, adding more flip-flops to the design.

3.6.7 Area

The baseline and SWIFT routers primarily differ in the following hardware components: tokens, lookahead signals with corresponding bypassing logic, buffers, VCs, and crossbar implementation. We estimate the standard-cell area contributions of each of these components, and compare these in Table 3.4. For the custom crossbar, we use the combined area of the RSD, switching devices and sense amplifier circuits as the metric to compare against the matrix crossbar's cell area. The total area of the SWIFT router is 25.7% smaller than the baseline router. This is the essential take-away of the SWIFT design: the 8% extra circuitry for tokens and bypassing, in turn results in a 11.2% reduction in buffer area and 49.3% reduction in control area (due to fewer VCs and corresponding smaller decoders and allocators) required to maintain the same peak band width, thereby reducing both area and power!

The SWIFT NoC also has some wiring overheads. The 23-bit *token* signals from the 3-hop neighborhood at each router add 7% extra wires per port compared to the 64-bit data path. The 14 lookahead bits at each port carry information that is normally included in data flits and so are not strictly an overhead[9]. Additionally, while Table 3.4 highlights that the active device area of the reduced swing custom crossbar is less than that of a synthesized design, differential signaling requires routing twice as many wires as well as potentially requiring an additional metal layer if

[9]The flit width can either be shrunk or packets can be sent using fewer flits, both not incorporated in our results, which will further enhance SWIFT's area or performance benefits over the baseline.

shielding is required for the application.

3.7 Related Work

The TILEPro64 [81] from Tilera (inspired by MITs RAW processor [77]) is a 64-core chip that uses five separate 8×8 mesh networks. One of these networks is used for transferring pre-defined static traffic, while the remaining four carry variable-length dynamic traffic, such as memory, I/O, and userlevel messages. The TRIPS [31] chip from UT Austin uses two networks, a 5×5 operand network to replace operand bypass and L1 Cache buses, and a 4×4 on-chip network (OCN) to replace traditional memory buses. The Intel TeraFLOPS [37] 80-core research chip uses an 8×10 network for memory traffic. Table I compares the design components for these three prototypes for their multiflit memory networks. The table also summarizes the design of our state-of-the-art BASELINE (t_r=3) NoC (designed for comparison purposes similar to UT TRIPS and Intel TeraFLOPS), which was described earlier in Section 3.2. The three prototypes used text book routers [22] with simple flow control algorithms, as their primary focus was on demonstrating a multicore chip with a packet-switched network beyond conventional buses and rings. In the SWIFT NoC project, we take a step further, and explore a more optimized network design, TFC [56], with reduced-swing circuits in the data-path. We simultaneously address network latency (buffer bypassing, one-cycle router), throughput (adaptive routing, buffer bypassing at all traffic levels using tokens and lookaheads), and power (buffer bypassing, low-swing interconnect circuits, clock gating). The SWIFT NoC optimizations can potentially enhance the simple networks of all these multicore prototypes.

3.8 Chapter Summary

In this chapter, we presented the microarchitecture of a 1-cycle router, and detailed our efforts in prototyping an instance of this design called SWIFT. We fabricated a 2×2 slice of a target 8×8 mesh NoC in 90nm. A lookahead-based router pipeline bypassing scheme, coupled with reduced-swing links in the crossbar and between routers, contribute to a 39% reduction in average network latency and 38% reduction in router power with uniform random traffic, compared to a state-of-the-art NoC with 3-cycle routers. Full-system simulations demonstrate a 26% reduction in runtime, which is within 2% of what an ideal NoC with 1-cycle routers and no contention can provide. The microarchitectural novelties of SWIFT can potentially be applied to any NoC router design.

In the rest of the chapters, we assume a SWIFT router-like 1-cycle router as our baseline. Since token-based adaptive routing does not gain us a lot in full-system performance, we henceforth use simple XY routing in our baseline, unless specified. We also assume full-swing circuits in the crossbar and links, both in our baseline as well as all proposed NoCs, to be able to use architectural power simulators Orion 2.0 [43] and DSENT [76] which do not model low-swing circuits. We will refer to this baseline design as the **BASELINE (t_r=1)** NoC in the rest of the thesis. In the next two chapters, we enhance this NoC to support 1-to-Many and Many-to-1 communication flows.

4 Single-cycle Per-hop NoC for 1-to-Many Traffic

Friends, Romans, countrymen, lend me your ears.
- William Shakespeare, *Julius Caesar, Act III, Scene II*

This chapter presents a NoC architecture with forking support at routers to efficiently deliver 1-to-Many (broadcast/multicast) traffic that is present in many shared memory cache coherence protocols. The techniques presented aim to reduce both per-hop contention $t_c(h)$ (via in-network forking and load-balanced distribution) and router delay t_r (to 1-cycle) in Equation 1.1 for 1-to-Many flows, unlike prior works which optimize one term at the cost of others.

4.1 Introduction

There are several different types of cache coherence protocols, each of which places different demands on the network connecting the cores. At one end of the cache coherence protocol design spectrum are broadcast-based protocols [61, 8, 75, 9]. These designs have the advantage of not requiring any directory storage, but have the limitation of increased network bandwidth demands because all requests and invalidates are broadcast. At the other end, full-state directory protocols [59, 60] track all sharers, reducing network demand by replacing broadcasts with precise unicasts and multicasts. However, the required storage increases area and energy costs as core counts scale. More scalable directory protocols [34, 15, 21, 1, 71, 58], including commercial designs like Intel's Quick Path Interconnect (QPI) protocols [1] and AMD's HyperTransport^{TM}Assist [21] incorporate partial directories to consume less storage than a full-state directory, and rely on a combination of broadcasts, multicasts, and direct requests to maintain coherence.

Section 2.2.1 in Chapter 2 we classified the communication patterns of protocols as 1-to-1, 1-to-M, and M-to-1 where M refers to multiple sources or destinations ($1 < M <= N$). 1-to-1 communication occurs in unicast requests/responses exchanged between cores. 1-to-M communication occurs in broadcasts and multicast requests [9, 1, 8, 75, 61]. M-to-1 communication occurs in acknowledgements [9, 1] or token collection [61, 71] in protocols to maintain ordering. In the on-chip domain, conventional wisdom dictates that 1-to-M and M-to-1 traffic should be avoided, assuming the on-chip network will not be able to handle such high bandwidth. Instead, the coherence mechanism involves serialized lookups through multiple caches and directories, adding latency to misses.

In this chapter and the next, we challenge this conventional wisdom and show that a network specifically designed to handle 1-to-M and M-to-1 traffic can approach the performance of an ideal network with 1-cycle routers, eliminating the need to avoid these patterns in the coherence protocol. In this chapter, we present Flow Across Network Over Uncongested Trees (FANOUT) [52], a set of optimizations that address inefficiencies in current state-of-the-art 1-to-M network designs. We propose a load-balanced routing algorithm called *Whirl*, a crossbar circuit called mXbar that forks flits at the similar energy/delay as unicasts, and a flow control technique for bypassing buffers; to realize single-cycle routers for 1-to-M flows. We also present a fabricated prototype of a 4x4 NoC with FANOUT routers at every hop, running at 1GHz [66].

Simulations with synthetic broadcast traffic show 61% lower latency, and 63% higher throughput than a state-of-the-art baseline NoC with multicasting support. Full-system simulations with the broadcast-intensive Token Coherence protocol show 10% reduction in application runtime, and 20% reduction in network energy, which are just 2.5% and 12% higher than the runtime and energy of a network with ideal 1-cycle multicast routers.

Section 4.2 of this chapter motivates our work by presenting relevant related work that form our baselines, and presents the characteristics of an ideal multicast network. Section 4.3 to Section 4.5 present the various features of FANOUT. Section 4.6 presents our fabricated chip. Section 4.7 presents our evaluations with synthetic and full-system traffic. Section 4.8 concludes.

4.2 Motivation

In Chapter 2 Section 2.2.1 we presented the breakdown of 1-to-1, 1-to-M and M-to-1 message flows for several multi-threaded workloads across HyperTransport [9] and Token Coherence [61]. For HyperTransport, 1-to-M requests form 14.3% of injected messages on average, while for Token Coherence, they form 52.4% of all injected messages, with M = 64 (i.e., full-chip broadcast) in both cases. There is clearly a need for NoCs that optimize for such traffic.

4.2.1 Baseline NoCs for 1-to-M flows

Most research in on-chip networks has concentrated only on optimizing 1-to-1 flows. 1-to-M flows are realized by sending M unicast packets from the source NIC. This approach causes heavy congestion at the link from the source NIC, creating hot spots. It also floods the network due to the bursty nature of these messages, loading some links M times over their capacity of 1 flit per cycle, leading to high contention and a dramatic rise in packet latency and corresponding penalties in throughput and energy. We will use the term **fork@nic** to refer to this baseline.

VCTM [41] identified the congestion problem for 1-to-M traffic with fork@nic designs, and there have been recent works [41, 26, 72, 79, 73] with solutions to mitigate it. While these differ in terms of routing algorithms, target systems, and the scale of M, in essence they propose routers with the ability to fork flits (i.e., a single multicast packet enters the network, and multiple flits are replicated and sent out of each output port towards their destinations at intermediate routers). We collectively term these works as **fork@rtr**.

4.2.2 Ideal Broadcast Mesh

Table 4.1: Theoretical Limits of a $k{\times}k$ mesh NoC for unicast and broadcast traffic.

Metric	**Unicasts**	**Broadcasts**
Average Hop Count ($H_{average}$)	$2(k+1)/3$	$(3k-1)/2$, for k even $(k-1)(3k+1)/2k$, for k odd
Channel Load on each bisection link ($L_{bisection}$)	$k{\times}R/4$	$k^2{\times}R/4$
Channel Load on each ejection link ($L_{ejection}$)	R	$k^2{\times}R$
Theoretical Latency Limit given by $H_{average}$	$2{\times}2(k+1)/3$	$2{\times}(3k-1)/2$, for k even $2{\times}(k-1)(3k+1)/2k$, for k odd
Theoretical Throughput Limit given by $\max\{L_{bisection}, L_{ejection}\}$	R, for $k <= 4$ $k{\times}R/4$, for $k > 4$	$k^2 \times R$

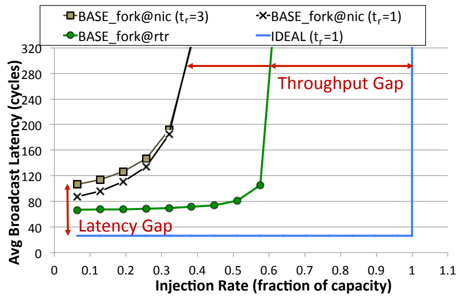
A mesh topology by itself imposes theoretical limits on minimum possible latency, and maximum possible throughput for different kinds of traffic. Here we derive and report the limits for uniform random unicast and broadcast traffic.

We assume a $k{\times}k$ mesh NoC injecting two kinds of traffic, unicast and broadcast. Specifically, each NIC injects flits into the network according to a Bernoulli process of rate R, to a random, uniformly distributed destination for unicasts, and from a random, uniformly distributed source to all nodes for broadcasts. All derived bounds are for a complete action: from initiation at the source NIC, till the flit is received at all destination NIC(s). For unicasts, these theoretical limits can be found in any textbook on NoCs [22]. For broadcast traffic, to the best of our knowledge, no prior theoretical analysis exists, and we describe our derivation next. Our results are reported in Table 4.1.

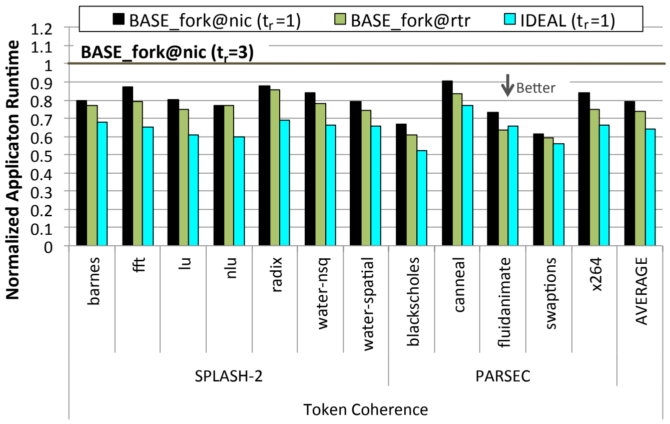
We derive the theoretical latency limit for received packets by averaging the hop delay from each source NIC to its furthest destination NIC. We assume a 1-cycle router + 1-cycle link at every hop.

We derive the theoretical throughput limit by analyzing the channel load across the ejection links[1] and bisection links[2], and observed that the maximum throughput for broadcast traffic is limited by the ejection links. This is unlike unicast traffic, where the bisection links limit throughput [22].

[1]The links from the router to the NIC.
[2]The links at the center of the mesh that partition the network into two halves.



(a) **Synthetic broadcast-only traffic.**



(b) **Full-system Runtime.**

Figure 4-1: Motivation: Gaps from the Ideal.

4.2.3 Gap from the Ideal

We compare fork@nic networks, with both 3-cycle and 1-cycle (similar to the one presented in Chapter 3) routers, and the fork@rtr network against a network with ideal 1-cycle multicast routers. We refer to these as **BASE_fork@nic (t_r=3)**, **BASE_fork@nic (t_r=1)**, **BASE_fork@rtr**, and **IDEAL (t_r=1)** respectively.

Figure 4-1(a) shows the latency and throughput gap between the baseline networks and the ideal (from Table 4.1) for uniform random synthetic broadcast-only traffic. We study the average latency for a broadcast to reach all destinations. We can see that fork@nic networks get saturated very early due to heavy contention, while fork@rtr networks are still far from the ideal in latency and throughput.

Figure 4-1(b) plots the full-system application runtime of systems running with these different networks. We study Token Coherence since that has 52% broadcasts. Compared to BASE_fork@nic (t_r=3), BASE_fork@nic (t_r=1) is 21% faster, and BASE_fork@rtr is a further 5% faster. However, BASE_fork@rtr is still 13% away from the IDEAL (t_r=1) NoC.

The goal of FANOUT is to bridge these gaps. To understand the reason for these gaps, we go back to the fundamental network latency Equation 1.1. The latency has a fixed (router) latency and a variable (contention) latency. In BASE_fork@nic (t_r=1), the former is reduced to 1, but the latter increases tremendously because of multiple unicasts being sent from the source to mimic a broadcast, and competing for the same set of resources along most of the route. In BASE_fork@rtr, contention delay is reduced, but it comes at the cost of an increase in router delay to $2+f$ cycles, where f is the number of ports the flit wishes to fork out from. The reasons for this increase in delay will be discussed, and addressed throughout this chapter.

Since Figure 4-1 shows that BASE_fork@rtr is better than BASE_fork@nic (t_r=1), *we use BASE_fork@rtr as the baseline design throughout the rest of the chapter.*

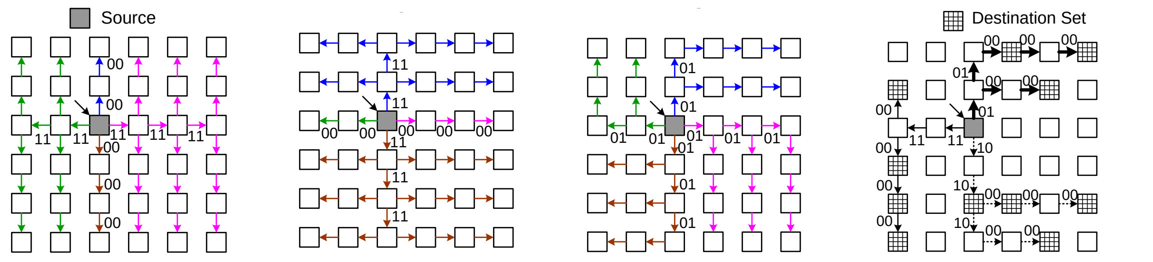
4.3 *Whirl*: Load-balanced 1-to-M Routing

4.3.1 Background

Multicast packets are typically routed in a *path-based* or *tree-based* manner. In path-based routing, a multicast packet is forwarded sequentially from one destination to the next; there is no forking/branching into other directions to reduce per-cycle bandwidth consumption, and packets are only forked out to the NIC at each destination router. For multicasts with many destinations, and for broadcasts, this leads to the packet snaking through the entire network. While this places the minimum load of 1 flit per cycle on each link, it results in extremely high latency for the destinations at the end of the snake, and is thus not a scalable solution.

Tree-based routing creates virtual multicast trees in the network, and are used in most prior works [41, 79, 72, 73, 26]. However, a major limitation of all these schemes is that their various multicast trees reduce to *one* tree in the presence of broadcasts or multicasts with many destinations (i.e., any node that broadcasts ends up using the same tree structure for distributing the broadcast). This is because as the broadcast moves through the network, it forks at intermediate routers based on *fixed* output port priorities to avoid duplicate reception of the same packet via alternate routes[3].

[3]For instance, in RPM [79], for destinations in the NE quadrant, the routers to the North of the source fork the broadcast flit to their East while those on the East of the source do not fork the flit. This ensures that only one copy of the flit is delivered to all nodes in the NE quadrants. Similar rules are enforced for other quadrants. But this always results in the YX-tree structure in Figure 4-2(b) for broadcasts.



(a) $Whirl_{W11-N00-E11-S00}$ (b) $Whirl_{W00-N11-E00-S11}$ (c) $Whirl_{W01-N01-E01-S01}$ (d) $Whirl_{W11-N01-E00-S10}$

Figure 4-2: Possible broadcast trees. *Whirl's algorithm: Packets fork into all four directions at the source router. By default, every packet continues straight in its current direction. In addition, forks at intermediate routers are encoded by LeftTurnBit, RightTurnBit, where left and right are relative to the direction of traversal. These bits are reset to 0 once a turn completes (hence, 0 is implicit on all unmarked arrows).*

The fixed broadcast trees created by VCTM [41], RPM [79] and bLBDR [72] are shown in Figures 4-2(a), 4-2(b) and 4-2(c) respectively, based on the output port priorities specified in the respective papers. The result is links are utilized in an unbalanced manner, lowering throughput. For broadcasts from uniformly distributed sources in an 8×8 mesh, we observed that VCTM [41] uses X-links 11% and Y-links 89% of the time, while RPM [79] uses X-links 89% and Y-links 11% of the time. Imbalanced loads on the different links increases contention, which adds delay and worsens throughput.

To the best of our knowledge, there has been no routing scheme that targets broadcasts/dense multicasts, and achieves ideal load balance.

4.3.2 *Whirl*

We propose *Whirl*: a tree-based routing scheme that (1) balances link loads for broadcasts and dense multicasts, (2) ensures non-duplicate packet reception, (3) is non-table-based, and (4) is deadlock-free.

Whirl parameterizes the entire space of possible broadcast trees, some of which are shown in Figure 4-2, and randomly selects one tree on each broadcast/multicast to balance the link loads. This is opposite to previous approaches, which build multicast trees from unicast paths and end up with one broadcast tree. For multicasts with few destinations, our approach and previous approaches yield similar results, but as the destinations increase, our approach outperforms previous approaches due to more path diversity, and thus lower contention.

At a high-level, *Whirl* randomly picks either an XY or a YX routing tree in every quadrant. The key challenge is to make sure that a flit does not reach the same router via two different trees. *Whirl* encodes the routing information for every broadcast/multicast packet in two bits: the LeftTurnBit (LTB) and the RightTurnBit (RTB). These tell the router whether the flit should turn[4] left or turn right *relative to* its current direction of motion. For instance, for a flit going West, left is South and right is North. The (LTB, RTB) pairs for each direction together create the global *Whirl* broadcast tree.

Choosing the ***Whirl*** broadcast tree

The global *Whirl* route for a packet is decided by the source NIC (sNIC, i.e., source routing). This is done not only to balance the load, but also to ensure non-duplicate and guaranteed reception of packets at all destinations' NICs; which is hard to support if the routers dynamically decide which route to take. The sNIC randomly chooses four bits: LTB_W, LTB_N, LTB_E, and LTB_S, which are the LeftTurnBits for each direction W, N, E, and S. The sNIC sends these four bits to the source router in the multicast packet.

Implementing forking using LTB and RTB

At the source router, the RTBs for each direction are computed from the four LTBs as follows: $RTB_S = \sim LTB_W$, $RTB_W = \sim LTB_N$, $RTB_N = \sim LTB_E$, $RTB_E = \sim LTB_S$. This rule enforces that duplicate copies of the same packet do not reach a router via different directions. For instance, in Figure 4-2(c), $RTB_N = 1$ and $LTB_E = 0$ to ensure non-duplicate delivery in the NE quadrant. The flit is then forked out of all four output ports, with each copy carrying the corresponding (LTB, RTB). At all further routers, the routing algorithm that is followed is shown in Figure 4-3. After

[4]A turn here implicitly implies a fork in a broadcast scenario as the flit also continues straight.

```
LTB: LeftTurnBit
RTB: RightTurnBit

At every router:
(1) fork into NIC
(2) continue STRAIGHT
(3) if: LTB is high then: fork LEFT
(4) if: RTB is high then: fork RIGHT
(5) clear (LTB,RTB) in flits that
    forked left/right.
```

Figure 4-3: Whirl pseudo-code

the flit turns once, no further turns are allowed, hence the (LTB, RTB) are reset to 0. This is done for simplicity, and to implement deadlock freedom, which we discuss later in this section.

Throughput characterization

Packets traversing a combination of *Whirl*'s 16 broadcast trees use all possible links that lie along the minimal routing path. For broadcast-only traffic from uniformly distributed sources, simulations showed 50% utilization on both the X and Y links, demonstrating ideal load balance.

Deadlock avoidance

Whirl allows all turns except U-turns, and thus requires a deadlock avoidance mechanism. We do not wish to restrict any turns and take away the ideal load balancing benefits of *Whirl*'s throughput discussed earlier. We thus apply conventional Virtual Channel[5] (VC) management to avoid deadlock, as shown in Figure 4-4. We partition the VCs into two sets, VC-a and VC-b. Packets are enforced to allocate only VC-a in the South direction, before they turn. They can allocate both VC-a and VC-b along the other directions, and after turning. Since all packets can only make one turn in *Whirl*, this restricts S-to-E and S-to-W turns within VC-b, implementing a

[5]Input buffers at routers are typically divided into multiple virtual channels to avoid packets going out of one port getting blocked by packets going out of another port.

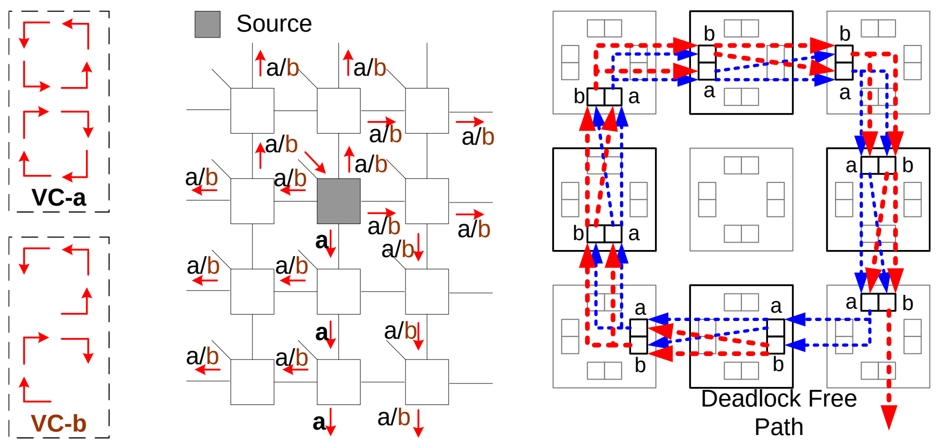


Figure 4-4: Deadlock avoidance by VC partitioning: VC-b implements a deadlock-free South-last turn model and acts as an escape VC.

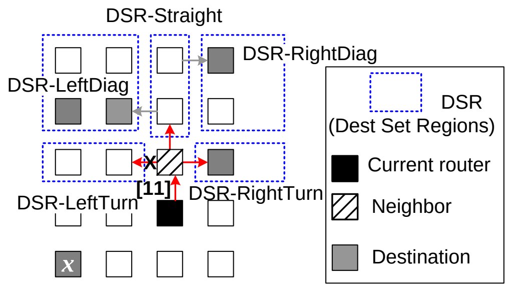
deadlock-free South-last turn model (Section 2.1.2). Because the multicast tree can be decomposed into unicast paths, VC-b acts like an escape VC [22][6]. Figure 4-4 shows an example scenario with all flits in VC-a in a circular dependency. However, VC-b will always have an escape path, and the flits in VC-a will eventually drain out via VC-b.

Another cause of deadlock in multicast networks is when two copies of the same flit take two alternate paths to reach the same destination. This can never occur in *Whirl* because of the LTB/RTB rules.

Point-to-Point Ordering

Multiple *Whirl* routes from the same sNIC can violate point-to-point ordering from source to destination. For coherence and other on-chip communication protocols that rely on this ordering, such as persistent requests in Token Coherence [61], sNICs statically assign only one of the *Whirl* trees, based on cache-block address, to all messages within an ordered virtual network/message class. Routers follow FIFO ordering for flits within an ordered virtual network, by using queueing arbiters for switch allocation, thereby guaranteeing point-to-point ordering.

[6]The escape VC [22] concept proves that as long as packets are allowed to allocate any VC, the sufficient condition to break deadlocks is to have one VC enforce a deadlock-free route while all other VCs can permit all turns.



(a) **Dest Set Regions (DSR) for multicast using Whirl.** *In this example, the north neighbor's DSR-LeftTurn and DSR-Straight are empty, while DSR-LeftDiag, DSR-RightDiag and DSR-RightTurn are non-empty. DSR-LeftTurn and DSR-RightTurn occupancy overrides the LTB and RTB values respectively, when deciding to turn, and thus the packet does not turn left. However, the packet continues straight even though DSR-Straight is empty, because DSR-LeftDiag and DSR-RightDiag are non-empty, and both LTB/RTB are high. The destination x is not in any of these DSRs, as it would be reached via some other router based on the global Whirl tree.*

```
(1) if :
      is_bcast or in DestSet
    then:
      fork into NIC
(2) if:
      is_bcast, or
      DSR-Straight is non-empty, or
      LTB is high and DSR-LeftDiag is non-empty, or
      RTB is high and DSR-RightDiag is non-empty
    then:
      continue STRAIGHT
(3) if:
      LTB is high and
      (is_bcast or DSR-LeftTurn is non-empty)
    then:
      fork LEFT
(4) if:
      RTB is high and
      (is_bcast or DSR-RightTurn is non-empty)
    then:
      fork RIGHT
(5) clear (LTB,RTB) in flits that forked left/right.
```

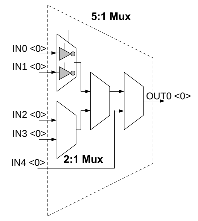
(b) **Whirl pseudo-code for multicasts.**

Figure 4-5: Whirl for multicasts.

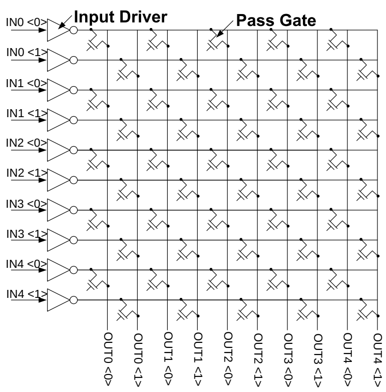
Pruning the tree for multicasts

For multicasts (in which not all NICs are in the destination set), flits need to carry their destination set with them, and *Whirl*'s algorithm described in Figure 4-3 can be modified such that flits do not continue/fork if no destination exists among the nodes reachable by that direction. As an example, Figure 4-2(d) sketches a *Whirl* broadcast tree trimmed for a multicast to 11 destinations.

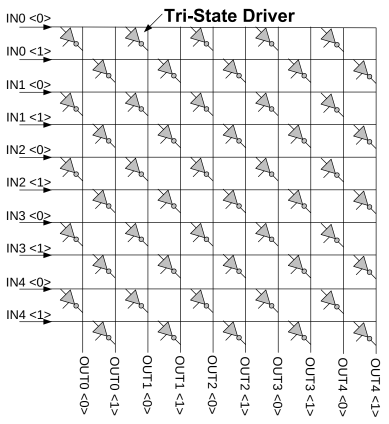
We assume that multicast flits carry a destination set bit-string, similar to that in RPM [79]. In our design, the destination bit-string gets divided into five Destination Set Regions (DSR) bit-strings, during the route computation, based on the position of the neighbor router for which the routing is being performed, as shown in Figure 4-5(a). These regions are called DSR-LeftTurn, DSR-LeftDiag, DSR-Straight, DSR-RightDiag and DSR-RightTurn. Unlike broadcasts, the decision to continue straight, turn left, and turn right depends not just on the LTB/RTB, but also on



(a) **Mux-crossbar (XBAR-A)**



(b) **Matrix-crossbar with pass-gate crosspoint (XBAR-B)**



(c) **Matrix-crossbar with tri-state crosspoint (XBAR-C) = mXbar**

Figure 4-6: Crossbar switch circuits.

the occupancy of each DSR, as highlighted in Figure 4-5(b). Note that the same destination node will lie in different DSRs for different neighbors of the same current router. This is not a problem, because the LTB/RTB rules described earlier will ensure that the destination is reachable from only one of the neighbors. For the same reason, bits in the destination-set bit-string do not have to be reset as the flit moves through the network, like in RPM [79], thereby simplifying the circuitry further.

4.4 mXbar: Router Microarchitecture for Forking

4.4.1 Background

Multicast routers fork flits out of multiple ports. This can be done either by (1) reading the same flit out of the buffer every cycle and sending it out of each output port one by one upon successful allocation, or by (2) reading the flit out of the buffer once and forking it within the crossbar.

The first approach adds serialization delay to multicast flits, increases buffer occupancy (thereby lowering throughput), and consumes high buffer read energy. However, it can use a simpler crossbar circuit that need not fork flits. VCTM [41] and

MRR [26] use this technique[7].

The second approach removes serialization delay, but requires a crossbar that performs forking. Samman et al. [73] and RPM [79] use mux-based crossbars to realize this. Mux-based PxP crossbars consist of a P:1 mux at each output port, as shown in Figure 4-6(a), and are the default designs generated from RTL synthesis. We call this XBAR-A. Because each input fans out to each of these muxes, this design can inherently fork flits out of multiple output ports. In contrast, a conventional matrix-crossbar is laid out as a regular matrix, and uses pass-gates at crosspoints [78] to implement the switching action, as shown in Figure 4-6(b). It relies on an input driver to drive (charge/discharge) *both* the horizontal and vertical wires of the crossbar. We call this XBAR-B. This design *cannot* efficiently fork flits because the driver cannot drive one horizontal *and* P vertical wires within a cycle[8]. The caveat, however, is energy. Crossbars are known to be one of the most power-consuming components of a router [37]. Each P:1 mux in XBAR-A is typically realized using a cascade of smaller 2:1 muxes, as shown in Figure 4-6(a), which increases energy consumption tremendously because many more transistors are used than in XBAR-B, as shown in Table 4.2. In addition, matrix-crossbar XBAR-B can segment wires [78] to drive only the required portion of the wires, saving more power.

We modeled these crossbars with five inputs/outputs in Orion 2.0 [43] at 45nm, targeting a 2GHz clock. Table 4.2 compares the delay/energy to perform a 1-to-M fork within a router. XBAR-A consumes 4.6X more energy than XBAR-B, and the corresponding router consumes 2.1X more energy even for a unicast (using $M = 1$ in the last column of Table 4.2), making it an impractical design to use. RTL-synthesized crossbars are not the only option in real designs; Intel's 80-core NoC [37] uses custom layouts for the crossbars to exploit regularity and reduce power.

[7]In fact, MRR does not use a crossbar. It rotates flits across buffers at different output ports.

[8]To fork flits, the input driver would have to be made about P times bigger, which would proportionally increase power consumption and become overkill for unicasts.

Table 4.2: Energy-Delay comparison for 5x5 128-bit crossbars, modeled using Orion 2.0 [43], at 45nm

Xbar	A	B	C
Type	Mux-based	Mat. + PassGate	Mat. + TriState
Transistors	240/bit	25/bit	150/bit
1-to-M Xbar Delay	1	M	1
1-to-M Xbar Energy	221×M fJ	48×M fJ	65×M fJ
1-to-M Router Energy	$E_{wr} + E_{rd} + M \times E_{xb}$ = 117 + 221×M fJ	$E_{wr} + M \times E_{rd} + M \times E_{xb}$ = 63 + 102×M fJ	$E_{wr} + E_{rd} + M \times E_{xb}$ = 117 + 65×M fJ

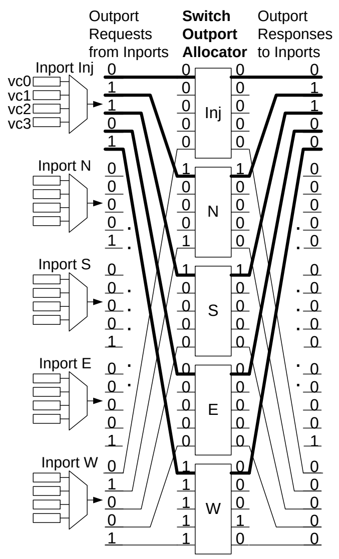
$E_{wr/rd}$ = Energy for buffer write/read
E_{xb} = Energy for xbar 1-to-1 traversal

4.4.2 mXbar: Multicast Crossbar

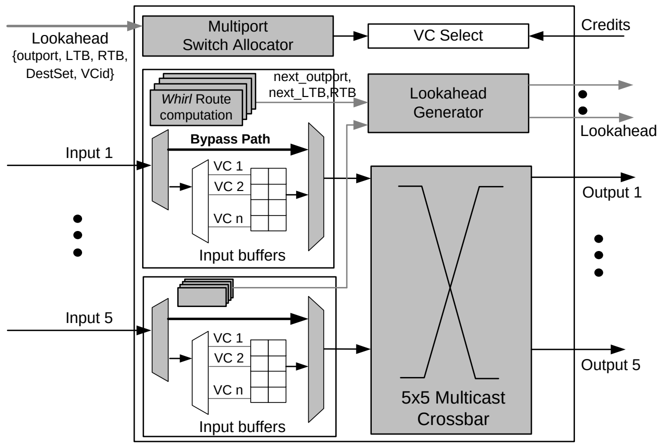
We propose a crossbar circuit, XBAR-C, that uses a matrix-crossbar layout but has tri-state drivers at crosspoints instead of pass-gates, as shown in Figure 4-6(c). The advantage of this design is that each output wire gets an independent driver, like XBAR-A, and can thus support forking of flits within a cycle. We thus trade the area advantage from XBAR-B by adding more transistors for higher drive-ability. The matrix-crossbar design still gives us the layout regularity and wire segmentation [78] advantages relative to the mux-based design. Table 4.2 shows that XBAR-C achieves a single-cycle delay like XBAR-A, while the router energy for forking flits with XBAR-C is 1.1X-0.8X the router energy of XBAR-B for 1-cast to 4-casts. XBAR-C is thus a practical design for forking flits, and we use it in the FANOUT router, referring to it as mXbar in the rest of the paper.

4.4.3 mSA: Multiport Switch Allocation

We show the design of our multiport switch allocator in Figure 4-7(a), which enables an input port to gain access to multiple output ports of the mXbar in the same cycle. In this example, inport *Inj* requests outports *N*, *S*, and *W*, and is granted *N* and *S*. At the end of mSA, the winner of an output port is granted a free VC for the next router from a queue of free VCs [55].



(a) **Multiport Switch Allocator.**



(b) **FANOUT Router Microarchitecture.** *Changes from baseline are shaded in grey.*

Figure 4-7: Single-cycle FANOUT Router.

4.5 Single-cycle FANOUT Router

4.5.1 Background

Single-cycle router pipelines have been proposed in the past [63, 57, 56] for unicast flows. In these designs, the router pipeline on the critical path reduces to just Switch Traversal (ST). The basic idea is to pre-allocate the crossbar switch before the actual flit arrives, to give it a direct access to the crossbar, thereby *bypassing* the buffering stage. We attempt to design such a pipeline for multicast flits. To the best of our knowledge, no prior research has attempted to extend buffer bypassing for multicasts, which is essential for meeting the delay/energy limits of an ideal broadcast network.

4.5.2 Pipeline Stages

We start by enumerating the various pipeline stages in the FANOUT router, and then start folding stages over each other to ultimately result in a single pipeline stage for multicast flits, as highlighted by Figure 4-8. This FANOUT pipeline is for the

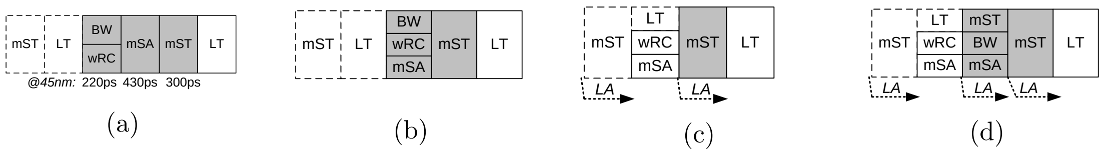


Figure 4-8: FANOUT pipeline optimizations to realize single-cycle multicast router: (a) 3-stage pipeline, (b) 2-stage pipeline, (c) 1-cycle pipeline (with bypass), (d) 2-stage pipeline (if bypass fails)
The flit pipeline at Router 2 is shaded in gray and that of preceding Router 1 is outlined with dashed lines. The stages are BW: buffer write, wRC: whirl route computation, mSA: multiport switch+VC allocation, mST: mXbar switch traversal, LT: link traversal, LA: lookahead.

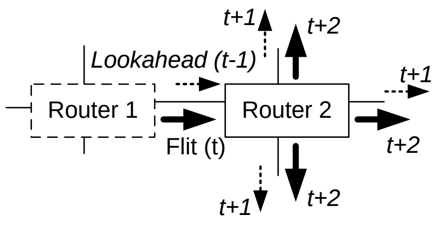


Figure 4-9: Traversal Example: Flit and Lookaheads.

message class[9]. Figure 4-7(b) shows the microarchitecture of the FANOUT router.

Step 1: Original pipeline (Figure 4-8(a)). The unoptimized FANOUT router pipeline is the same as a baseline fork@rtr design, though the actual components (routing algorithm, switch allocation algorithm, and crossbar circuit) differ. At Router 1, the flit goes through the switch (mST) and link (LT) to arrive at Router 2. It gets buffered (BW), performs routing (wRC) in parallel, then places a request for multiple ports of the switch (mSA), forks through the switch (mST), and traverses the link (LT) to the next router. The critical path delays for each are shown in Figure 4-8(a), obtained from RTL implementation of the FANOUT router in 45nm and synthesizing for a 2GHz clock.

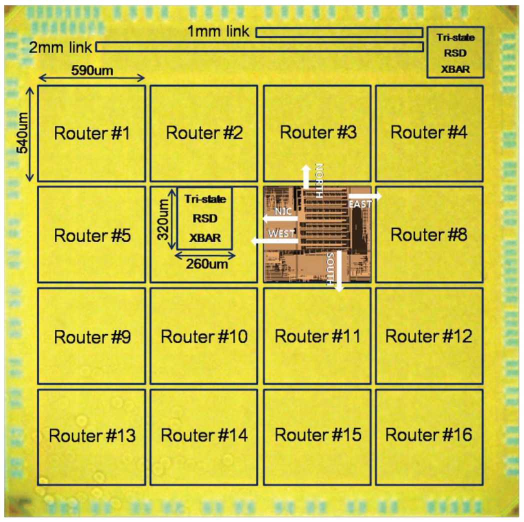
Step 2: Lookahead routing (folding mSA and wRC) (Figure 4-8(b)). The multiport switch allocation (mSA) cannot be performed until the output port requests for the flits are known, which are only available after *Whirl* route computation (wRC). We leverage lookahead routing [27] and perform wRC one hop in advance (i.e., the

[9]The router buffers are partitioned into separate virtual message classes such as requests, responses, etc. to avoid protocol-level deadlocks, as explained in Section 2.2.1.

wRC at the current router determines the output ports at the *next* router, allowing an incoming flit to place a request for the switch as soon as it arrives). Thus BW, wRC, and mSA can all be done in parallel. However, performing wRC for the *next* router is not as trivial as in a unicast case because multicast flits could have *multiple* next routers if they are forking at the current router. To perform routing one hop in advance, the current router needs to perform wRC for *all* output ports out of which the flit will fork. Thus in a 5x5 router, every input port needs to maintain four wRC blocks, one for each output port assuming no u-turns. The power and area overhead for these 20 *Whirl* blocks was found to be less than 1% of that of the total router because it is very simple combinational logic (Figure 4-3). The output port request generated by the *Whirl* block for output port A is embedded in the corresponding flit going out of port A.

Step 3: Bypassing (wRC and mSA before flit arrival) (Figure 4-8(c)). We can shrink the flit pipeline at the router to one cycle if we perform wRC and mSA before the flit arrives. To do so, we must examine what information is required by wRC and mSA. wRC needs to know the output ports out of which the flit will fork (5-bit vector) to activate the corresponding *Whirl* blocks at the input port. It also needs to know the 2-bit 'LTB,RTB' to determine the route. mSA needs to know only the output ports request. To realize this pipeline, the flit at Router 1 sends these 7 bits as a *lookahead* (LA) to Router 2, while it traverses the mXbar (mST) at Router 1. This enables Router 2 to perform wRC and mSA while the flit performs link traversal. If mSA at Router 2 is successful in granting all output ports to the flit, it does not get buffered and uses the single-cycle pipeline in Figure 4-8(c). This allows FANOUT to eliminate the buffer write and read energy, shown earlier in Table 4.2, from the router traversal. The mXbar is critical for achieving this one-cycle pipeline; otherwise a multicast flit will be forced to get buffered and spend multiple cycles to go out one by one. If mSA grants only a subset (or none) of the the ports, the flit gets buffered and starts mSA for the remaining ports, as shown in Figure 4-8(d).

A flit that performs mST (either via bypassing or from the buffers) needs to send lookaheads out of all output ports that it will fork out from, as shown in Figure 4-9.



Flit Size	64 bits
Request Packet Size	1 flit *(coherence requests and acknowledges)*
Response Packet Size	5 flits *(cache data)*
Router Microarchitecture	10 X 64b latches per port (6VCs over 2MCs)
Bypass Router-and-link Latency	1 cycle
Operating Frequency	1GHz
Power Supply Voltage	1.1V and 0.8V
Technology	45nm SOI CMOS

Figure 4-10: Die Photo and overview of fabricated 4×4 FANOUT NoC

These lookahead bits are ready at the end of wRC and mSA.

Ideal 1-to-M Traversal. In summary, *Whirl* sets up load-balanced paths for multicast flits to lower congestion, and the mXbar and lookaheads together allow flits to perform single-cycle forking at routers without getting buffered, thereby incurring only switch and wire (mST, LT) delay/energy from the source to all destinations.

4.6 Prototype Chip

In this section, we present our prototype chip demonstrating a single-cycle FANOUT router, embedded in a 4×4 NoC in 45nm SOI, running at 1GHz [66]. We do not implement *Whirl* in the chip since its throughput is similar to that of an XY-tree in a small 4×4 mesh.

The FANOUT chip tapeout was joint work with other students Sunghyun Park, Chia-Hsin Owen Chen, and Bhavya Daya from MIT. I designed the architecture and RTL, Sunghyun created the custom layout of the mXbar with low-swing drivers and receivers, Owen handled full-chip integration, and Bhavya performed spice simulations to validate the extracted netlists.

Figure 4-10 presents the die photo, and overview of our chip. The router has a measured critical path of 961 ps. We incorporate on-chip traffic generators capable

Table 4.3: Network Parameters.

Topology	8×8 mesh
Router Ports	5
Ctrl VCs/port	12, 1-flit deep
Data VCs/port	3, 3-flit deep
Flit size	128 bits
Link length	1mm

Table 4.4: Traffic Parameters.

Synthetic Traffic	
Unicast	Uniform Random Bit-Complement Tornado, Hotspot
Multicast % total traffic	5%, 20%, 60%, 100%
Multicast destination set	DEST_ALL: 2-64 random, DEST_FEW: 2-16 random DEST_MANY: 48-64 random
Full-System Traffic	
Processors	64 in-order SPARC
L1 Caches	Private 32 kB I&D
L2 Caches	Private 1MB per core
Coherence Protocol	(1) HyperTransport [9] (HT) (2) Token Coherence [61] (TC)
DRAM Latency	70ns

of generating both broadcast-only and mixed (33% broadcast request, 33% unicast request, and 33% unicast response) traffic. The chip meets the theoretical latency limits presented in Table 4.1. For broadcast-only and mixed traffic, the chip delivers a throughput of 932 Gbps and 892 Gbps respectively, which is 91% and 87% of the theoretical limit.

4.7 Evaluations

In this section, we evaluate FANOUT with both synthetic traffic, as well as full-system traffic. Table 4.3 lists our network parameters. The number of VCs in all configurations is set by the buffer turnaround time (Section 2.1.3) within the request and response message classes in the baseline to improve its performance. FANOUT achieves close to its peak performance with 2/3rd the number of VCs. Table 4.4 describes our synthetic and full-system traffic scenario. We target a 45nm technology

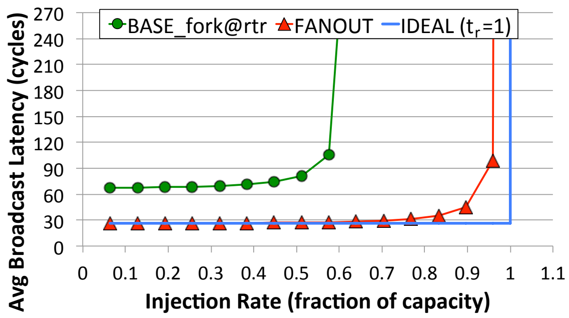


Figure 4-11: Uniform Random Broadcast Traffic

node with a VDD of 1.0V and a frequency of 2GHz.

4.7.1 Baseline and Ideal Network

We model the *BASE_fork@rtr* design, similar to VCTM [41], bLBDR [72], MRR [26], and RPM [79], with routers forking flits as they move towards their destination. The baseline broadcast follows an XY-tree routing algorithm, as explained in Section 4.3. The pass-gate matrix-crossbar from Figure 4-6(b) is used in the baseline for its low power and area.

The *IDEAL (t_r=1)* design models an ideal contention-less network with 1-cycle routers at each hop, as described in Section 2.3.1.

4.7.2 Network-only Synthetic Traffic Simulation

Limit Study with Broadcast Traffic

We start by studying FANOUT in the presence of only broadcasts, and characterize it against the ideal broadcast mesh metrics derived in Section 4.2.2. We inject a synthetic broadcast traffic pattern where uniformly-random sources inject single-flit broadcast packets into the network, at a specified injection rate. The metric we use for evaluation of latency is *broadcast latency* which we define to be the *latency between generation of a broadcast packet at a network interface, to the receipt of the tail flit of the last copy at the destination network interfaces.* Saturation throughput is the injection rate at which the average latency reaches 3-times the low-load latency.

Latency and Throughput. Figure 4-11 shows the average broadcast latency as a function of injection rate for FANOUT_whirl-mxbar-bypass - FANOUT with *Whirl* routing, multicast crossbar and multicast bypassing - compared to the baseline. The IDEAL (t_r=1) line is calculated from Table 4.1 by setting k=8. We observe that FANOUT_whirl-mxbar-bypass has 60.6% lower low-load latency, and 62.7% higher throughput than BASE_fork@rtr. *Whirl* by itself results in 22.2% improvement in throughput, multicast bypassing by itself[10] results in 37.0% reduction in low-load latency, and 22.2% improvement in throughput, and the mXbar by itself results in a 24.2% lower low-load latency, and 62.7% higher throughput. These are not shown in Figure 4-11 for clarity, but will be explored in detail later.

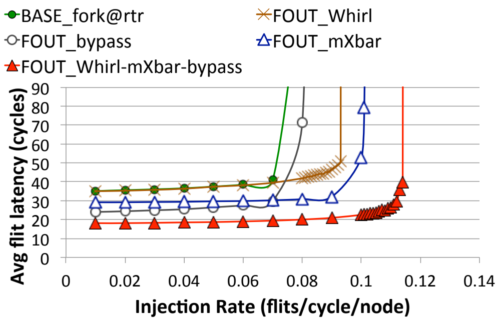
Energy. Near saturation, the energy savings of FANOUT over BASE_fork@rtr is 80.1% in buffer read/write energy (for 8 buffers per port in both networks), and 11.6% overall.

In summary, with worst case traffic (100% broadcasts), FANOUT's latency is 5% off ideal on average prior to network saturation, attains 96% throughput of the ideal, with energy consumption just 9% above ideal.

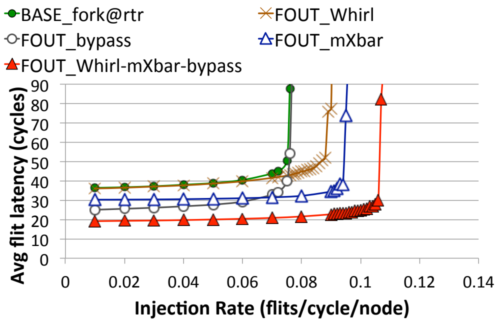
FANOUT with mix of unicast and multicast traffic

We evaluate the impact of FANOUT with multicast traffic in the presence of various kinds of unicast traffic (uniform random, tornado, bit-complement and hot-spot). We discuss the results for a network with 20% multicast traffic with number of destinations varying randomly from 2-64 at each injection (DEST_ALL from Table 4.4). For uniform-random (Figure 4-12(a)), *Whirl* helps improve throughput by 17.5%, mXbar reduces low-load latency by 18% and improves throughput by 26.3%, while multicast bypassing reduces low-load latency by 31.4%. Combining all three techniques results in a low-load latency reduction of 49% and throughput improvement of 43.7%. A similar trend is observed in bit-complement (Figure 4-12(b)). In tornado (Fig-

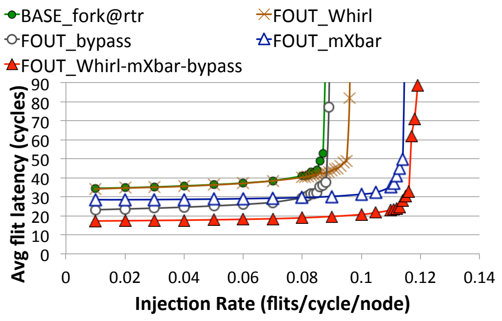
[10]Multicast bypassing by itself, without the mXbar, means that multicast flits send out lookaheads to try and preset the next crossbar, but only one output port can be requested at a time; so a copy of the flit is also retained to arbitrate for the remaining output ports in subsequent cycles. A 1-cycle FANOUT router can only be realized when both mXbar and multicast bypassing are enabled.



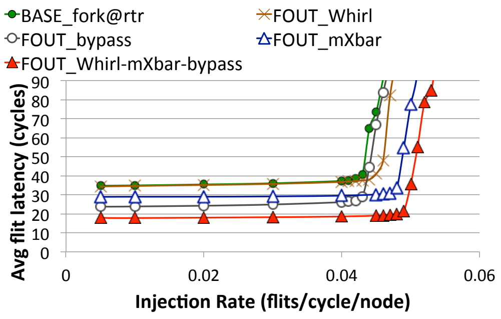
(a) **Uniform Random Unicast**



(b) **Bit-Complement Unicast**



(c) **Tornado Unicast**



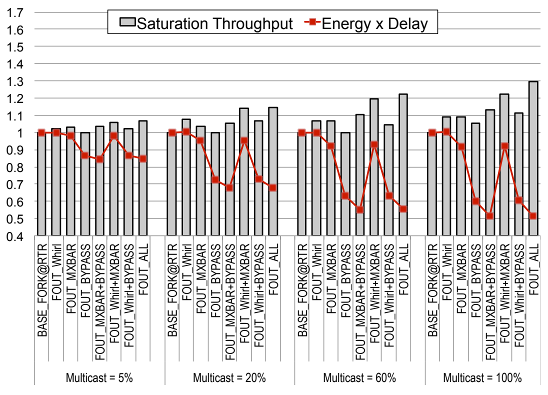
(d) **Hot-Spot Unicast**

Figure 4-12: Performance with 80% unicasts + 20% DEST_ALL multicasts.

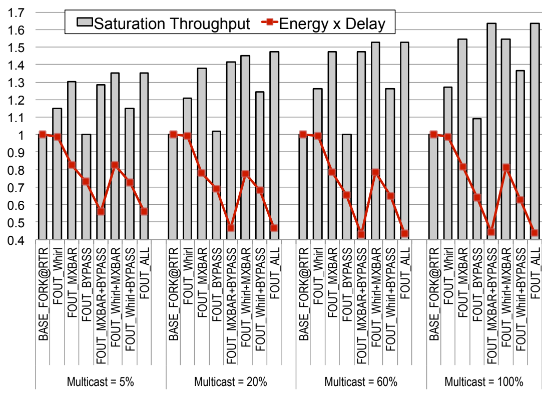
ure 4-12(c)), however, mXbar improves throughput by 30.7%, but adding *Whirl* and multicast bypassing to it do not improve throughput further like in uniform-random and bit-complement. This is because tornado traffic has unicast flits traveling continuously only on the X links. Thus load-balancing by *Whirl* ultimately gets limited by the highly imbalanced unicast traffic. An extreme case of this phenomenon is observed in Hot-Spot traffic[11] (Figure 4-12(d)). FANOUT enables 49% lower low-load latency, but is not able to push throughput by greater than 15% since the highly contended and imbalanced Y-links near the hot-spot nodes limit network saturation (since unicast traffic uses XY routing).

In summary, *Whirl* and mXbar help improve throughput, while mXbar and multicast bypassing help lower the latency, as highlighted in Figure 4-12. Since FANOUT enables each technique to gel with the other, combining them results in up to 49% reduction in latency, and 44% higher throughput, due to efficient and faster use of

[11]We selected four hot-spot nodes in the network, and all unicast traffic is directed to one of them randomly.



(a) **DEST_FEW (2-16) Multicasts + Uniform Random Unicast**



(b) **DEST_MANY (48-64) Multicasts + Uniform Random Unicast**

Figure 4-13: Normalized Saturation Throughput and Energy-Delay-Products for FANOUT's components.

links, unless adversarial unicast traffic limits the overall network.

Breakdown of impact of *Whirl*, mXbar and multicast bypass

Next we evaluate the impact of each component of FANOUT on performance and power as a function of the amount of multicast traffic in the network, and the size of the destination sets. Figure 4-13 plots the network saturation-throughput, and the Energy-Delay Product (EDP) at low-loads, for two kinds of destination sets: DEST_MANY (48-64 destinations randomly chosen) and DEST_FEW (2-16 destinations randomly chosen), and sweeps through the percentage of multicasts in the network. The unicast traffic is uniform-random in all cases. We demonstrate the impact of each component of FANOUT based on these results. FANOUT_*Whirl* refers to the BASE_fork@rtr network with *Whirl* routing. FANOUT_mXbar refers to the BASE_fork@rtr network with a multicast crossbar, but using the baseline multicast tree, and so on. FANOUT_ALL has all three techniques. We can see that in both traffic conditions, there is a consistent reduction in EDP due to FANOUT. For DEST_MANY, for which FANOUT is primarily intended, FANOUT's components lead to 40-60% higher network throughput, and upto 56% lower EDP.

Whirl. For DEST_FEW, with 5% multicasts, *Whirl*'s performance is very comparable to BASE_fork@rtr. This is expected because for such a configuration, both

Whirl and BASE_fork@rtr create routes that match the destination locations. *Whirl* wins slightly because conflicts are broken by random LTB/RTB choices in *Whirl* while BASE_fork@rtr uses fixed priorities. *Whirl* starts improving throughput as the percentage of multicasts increase. The real benefits of *Whirl* can be seen in the DEST_MANY where it gives 18-25% improvement by itself, and upto 60% in conjunction with FANOUT's other optimizations.

Whirl's EDP is similar to the fork@rtr, because at low-loads, at which the EDP was calculated, both *Whirl* and fork@rtr traverse similar routes, incurring similar number of buffer writes/reads and crossbar/link traversals.

mXbar. For DEST_FEW, the mXbar does not show a huge benefit, and offers throughput improvements similar to *Whirl*. The reason is that the percentage of unicast dominate over multicasts, so the mXbar does not have much work to do in terms of forking flits. As the percentage of multicasts increase, *Whirl*+mXbar push the throughput by 20-30%, and lower EDP by 35-50%. For DEST_MANY, however, mXbar steals the show completely. In almost all cases, it single-handedly pushes the throughput to within 10% of the maximum achieved by all three techniques together. However, as discussed earlier in Section 4.4, the mXbar has higher energy/bit/1-to-1 traversal, and was found to consume higher power than the baseline. But the reduction in latency due to mXbar offsets that and leads to 17-22% lower EDP than the baseline.

Multicast Buffer Bypassing. Multicast bypassing does not offer any significant throughput improvement by itself, in both DEST_FEW and DEST_MANY. This might seem contradictory to all previous works on unicast buffer bypassing [57, 56]. The reason for this is that bypassing helps increase throughput by enabling faster turnaround of buffer usage. For multicasts however, the buffer cannot be freed until all copies of the flit leave! Buffers can only be bypassed at routers which are not forking flits along the multicast route. But multicast bypassing by itself does have delay benefits, because it enables flits to proceed to the crossbar as soon as they enters, while a copy is also retained at the buffers. This enables a 20-40% reduction in EDP even in the DEST_FEW case. When multicast bypassing is combined with *Whirl*, or mXbar, more of its benefits come to light. With *Whirl* enabling more load balanced routing, the chances of bypassing routers outside of the destination set increases due to lower contention. With mXbar, the biggest benefit of bypassing is in terms of energy. mXbar coupled with bypassing enables huge savings in buffer read/write energy. It also allows faster recycling of buffers, and better link utilization, all of which push throughput by up to 60% in some cases.

In summary, for networks with few multicasts, and small destination sets, *Whirl* and mXbar provide similar performance improvements. Since *Whirl* adds minimum overhead to the router, it would be a better solution than re-designing the crossbar to support multicasts. For dense multicasts/broadcasts, however, the mXbar is critical for good performance. In this case, latency and power can be saved further by adding multicast bypassing.

4.7.3 Full-system Simulations

Application Runtime

We start by comparing the performance benefits of FANOUT with all its routing, flow control, and microarchitecture optimizations included, against the IDEAL (t_r=1) NoC. Figure 4-14 shows the normalized full-system application runtime with FANOUT for both protocols. With FANOUT, HT shows 10% improvement, while TC shows 9.7% improvement, on average. Compared to IDEAL (t_r=1), FANOUT is off by about 2.5% compared to the IDEAL for TC. For HT, however, FANOUT is off by 9.5%.

We also plot a second ideal, called IDEAL_1-to-M (t_r=1), which models ideal 1-cycle per-hop contentionless traversals for the broadcasts, while the rest of the messages go through the baseline network. We can see that FANOUT is less than 3% off this ideal for HT. This means that FANOUT by itself is successful in achieving near single-cycle per-hop traversals for the 1-to-M flows. The remaining gap from the IDEAL (t_r=1) network is because of the M-to-1 flows which will be addressed in Chapter 5.

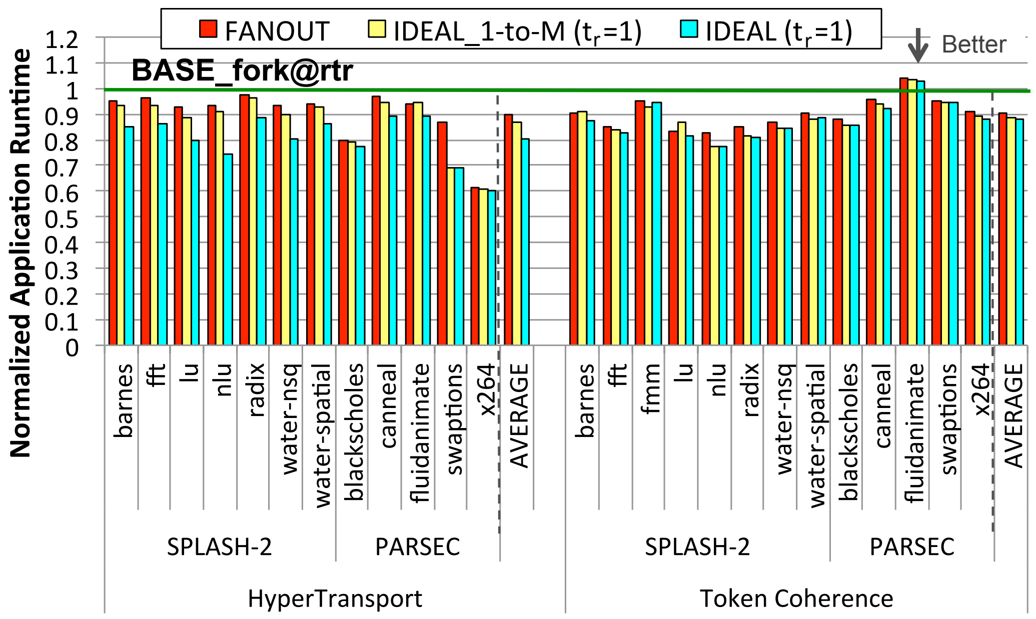


Figure 4-14: Full-system Application Runtime.

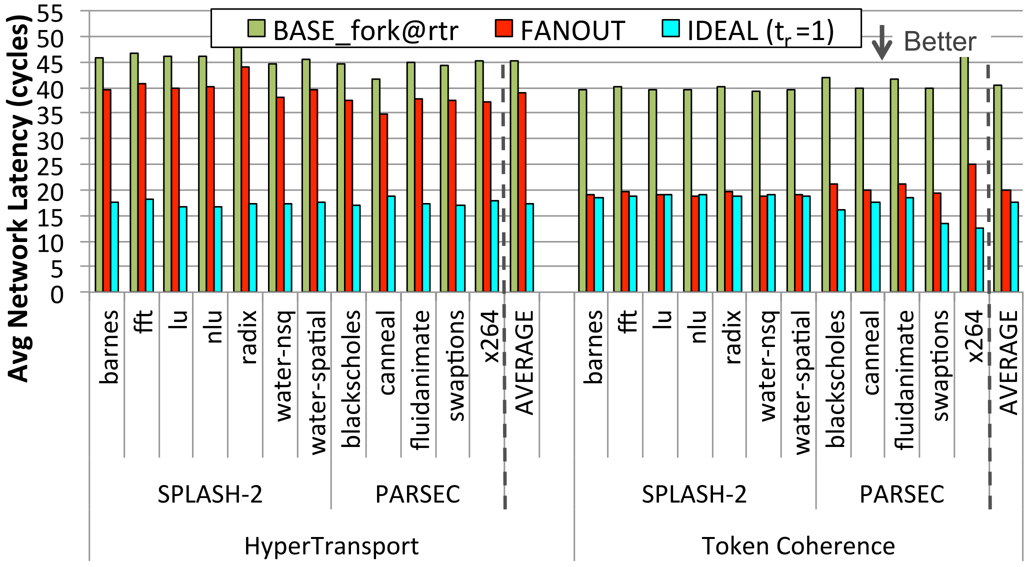


Figure 4-15: Average Network Latency.

Network Latency

The full-system runtime behavior seen earlier can be understood by looking at the network latency impact in Figure 4-15. We plot the average flit latency, which includes both broadcast flits (that FANOUT targets), and other unicast flits that are part of the protocol. The FANOUT design reduces average network latency by about 14% for

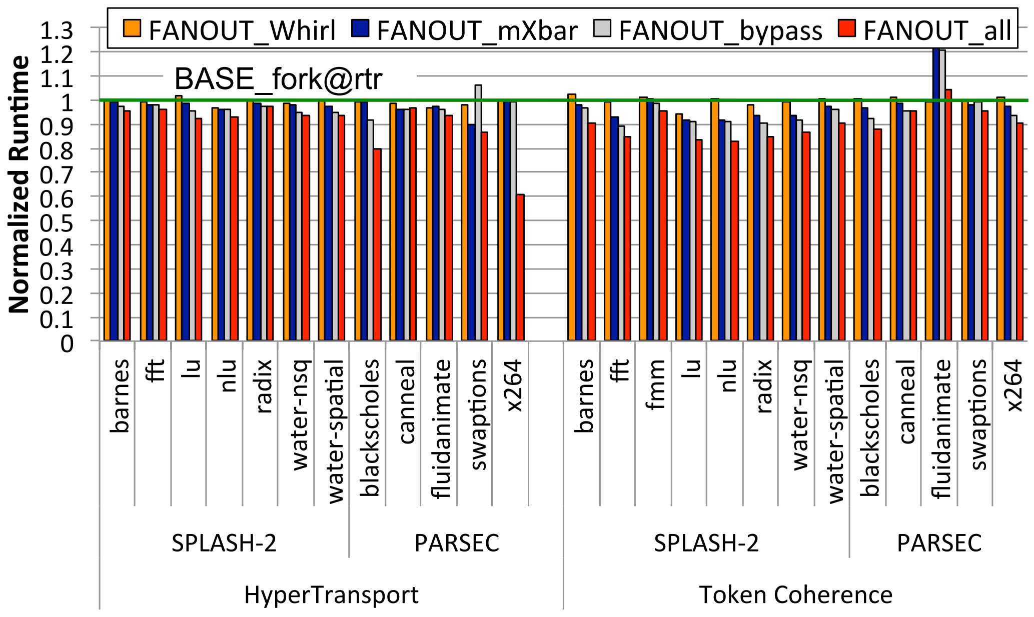
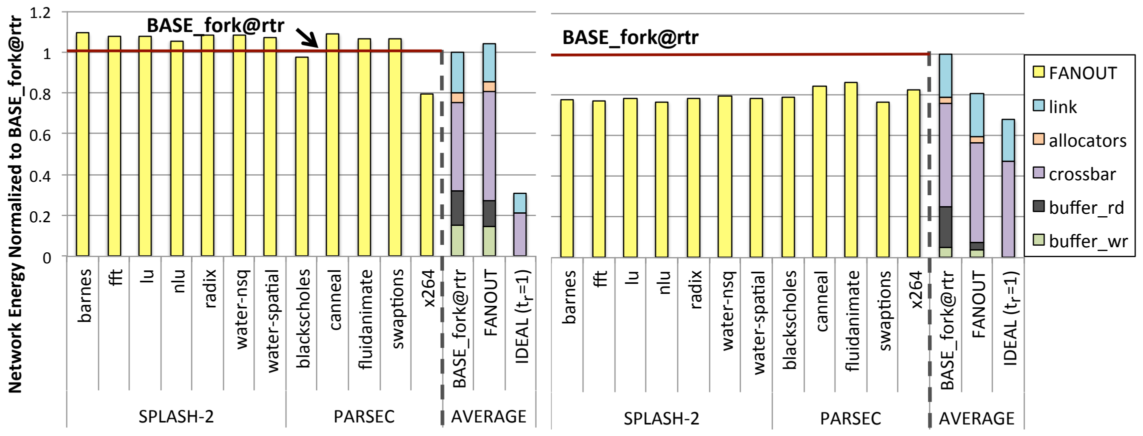


Figure 4-16: Impact of routing and flow control components.

HT and 50% for TC. This difference can be understood by looking at the breakdown of 1-to-M messages in the protocols. Broadcasts form 45-55% messages in TC, as opposed to 11-17% in HT. The average latency of broadcast packets was observed to go down by 40% on average in both HT and TC with FANOUT, consistent with the results observed earlier in Section 4.7.2. However, the dominance of bursty ACKs in HT increases its average latency across all packets, leading to an average of 2.3 cycles more per-hop compared to the IDEAL (t_r=1). For TC, FANOUT reduces the average network latency to about 20 cycles, which is just 0.3-cycles per-hop on average more than the IDEAL (t_r=1).

Impact of components of FANOUT

In Figure 4-16, we show the effect on full-system runtime of stand-alone components of FANOUT (*Whirl*, mXbar, bypass). *Whirl* shows 4% runtime savings for *nlu* and *fluidanimate* in HT and 6% for *lu* in TC. These are applications that have 1.3-2X higher injection rates than the others, and thus benefit from a load-balanced network. For the other applications, the XY-tree of the baseline does as well as *Whirl.* mXbar shows about 5-10% runtime reduction in TC for most benchmarks. In HT, *nlu,*



(a) **HyperTransport.** (b) **Token Coherence.**

Figure 4-17: Network Energy.

canneal and *swaptions* benefit from mXbar but other benchmarks do not because the performance starts getting limited by the ACKs. Buffer bypassing by itself shows 3-4% runtime improvement in average in both HT and TC. Without the mXbar, flits need to get buffered to fork out of the router, and buffers are bypassed only at those routers where no forks need to occur.

When all three components of FANOUT come together, flits follow load-balanced paths, fork through the mXbar, and avoid getting buffered, resulting in a 10% runtime improvement on average, which is higher than what any one technique provides.

Network Energy

Figure 4-17 shows the breakdown of network energy with FANOUT for HT and TC, normalized to BASE_fork@rtr.

For HT, the overall network energy actually goes up by 4.3% with FANOUT. The reason for this increase can be seen from the relative breakdown of buffer and crossbar energy. Buffer reads do go down with FANOUT (due to the single-cycle pipeline enabled by the mXbar), but only by 3.9% because the buffer accesses are dominated by the ACKs in HT. The crossbar energy, meanwhile, goes up by 10.1% because the mXbar consumes more energy than the baseline crossbar for unicasts, as explained earlier in Table 4.2. Compared to the IDEAL in energy[12], both BASE_fork@rtr and FANOUT are off by about 70%.

For TC, the story is more optimistic. The dominance of broadcasts allows FANOUT to provide a 19.7% reduction in energy, primarily due to a reduction in buffer reads (due to the mXbar) and buffer writes (due to multicast bypassing). Compared to the IDEAL, while BASE_fork@rtr is off by 32%, FANOUT reduces the gap to 12%.

4.8 Chapter Summary

In this chapter, we presented FANOUT, a collection of optimizations for efficient multicast support within NoC routers to allow 1-to-M flows to approach their ideal energy-delay-throughput characteristics. FANOUT includes a load-balanced routing algorithm for multicasts called *Whirl*, a crossbar circuit called mXbar that forks flits at the similar delay/energy as unicasts, and a single-cycle multicast router. A prototype chip in 45nm SOI validates the microarchitecture and flow control.

Evaluations with synthetic stress-tests highlight that FANOUT is able to achieve near-ideal single-cycle per-hop performance for 1-to-M flows. With full-system simulations, this is reiterated for Token Coherence. But for HyperTransport there is still a 9.5% gap with the runtime and a 70% gap with the energy of the IDEAL 1-cycle per-hop NoC, exposing the criticality of addressing the M-to-1 flow of acknowledgements in this protocol. The next chapter proposes in-network aggregation support for these ACKs. Together, the current chapter and the next form a solution to efficiently manage collective communication flows on-chip.

[12]The energy consumed in the ideal networks is just wire/data-path energy, i.e., ST and LT.

5 Single-cycle Per-hop NoC for Many-to-1 Traffic

Coming together is the beginning.
Keeping together is progress.
Working together is success.

- Henry Ford

This chapter presents a NoC architecture with aggregation (reduction) support at routers to efficiently deliver Many-to-1 (ACKs) traffic that is present in many shared memory cache coherence protocols. The techniques presented aim to reduce per-hop contention $t_c(h)$ (by aggregating flits at routers) and router delay t_r (to 1-cycle) in Equation 1.1 for Many-to-1 flows.

5.1 Introduction

We continue with the goal set in Chapter 4 to design a NoC to efficiently handle bursts of 1-to-M and M-to-1 traffic to allow coherence protocols to scale. Figure 2-14 plotted the breakdown of 1-to-1, 1-to-M and M-to-1 message flows for several multi-threaded workloads across HyperTransport [9] and Token Coherence [61]. For HyperTransport, 1-to-M requests and M-to-1 responses form 14.3% and 14.1% of injected messages on average, respectively, Token Coherence reduces M-to-1 traffic to 2%, with M = 12 on average, at the cost of a higher percentage (52.4%) of 1-to-M traffic (M = 64).

There has been no prior work, to the best of our knowledge, to deal with M-to-1 flows in a NoC. M-to-1 flows result in M unicast packets being received by the destination NIC. This creates heavy congestion at the link to the destination NIC, creating hot-spots. Moreover, the bursty nature of these flows (since all caches are likely to ACK a broadcast request at around the same time) floods the network loading some links M times over their capacity of 1 flit per cycle, leading to a dramatic rise in latency and corresponding penalties in throughput and energy.

In addition to coherence, M-to-1 flows occur in barrier synchronization [28], the reduce phase of MapReduce [23], and so on. Previous work has focused on specialized solutions for accelerating barrier messages [28, 5, 54, 80, 18] by relying on an ordered FAT-tree topology in the off-chip domain [28] and adding additional wires and registers to track barriers in the on-chip domain [5, 54, 80, 18]. Unfortunately, none of these solutions are generic enough to be applied for other M-to-1 traffic across an unordered, distributed on-chip network topology such as a mesh that is commonly used in multicore chips [37, 81].

Most M-to-1 flows described so far comprise single-flit responses acknowledging the receipt of some request or signaling the end of some transaction. This provides opportunity to perform a commutative aggregation of these responses within the network into one flit carrying a count of the number of responses it represents. In this work, we propose Flow AggregatioN In-Network (FANIN) [52] to perform such a reduction of M-to-1 traffic in a distributed manner. We add additional optimizations for synchronized routing (*rWhirl*) and a smart flow control for waiting at routers, to realize single-cycle routers ($t_r = 1$) for M-to-1 flows.

The motivation for FANIN stems from Figure 4-14 in Chapter 4 where we plot the runtime of the FANOUT NoC, and compare it with the runtime of a NoC with ideal 1-cycle routers at every hop. For HyperTransport, FANOUT is 9.5% away from the ideal, despite using 1-cycle routers for both unicast and multicast traffic, because of heavy contention at every hop ($t_c(h)$) due to M-to-1 ACKs. Token Coherence uses only 2% of M-to-1 flows and is thus close to the ideal with FANOUT itself.

The rest of the chapter is organized as follows. Section 5.2 discusses the background and related work. Section 5.3 presents a walk-through example of FANIN. Section 5.4 presents a load-balanced routing algorithm for M-to-1 flows. Section 5.5

describes FANIN's aggregation methodology within each router. Section 5.6 presents the pipeline of a FANIN router, and optimizations to realize a single-cycle aggregation router. Section 5.7 shows our evaluation results, and Section 5.8 concludes.

5.2 Background and Related Work

M-to-1 communication flow occurs in shared memory coherence protocols, in the form of acknowledgements [9, 1] or tokens [61, 71], and in message passing domains, such as barrier synchronization [28, 18, 5, 80]. There has been no on-chip solution to tackle the former, to the best of our knowledge. In the past, MIMD machines like IBM RP3 [68] and NYU Ultracomputer [30] used network switches to combine memory requests (loads/stores/fetch-and-add) to the same memory location and added extra buffers to track the responses. More recently, aggregation via the network fabric has been researched for implementing barrier synchronization [28, 5, 54, 80, 18]. These proposals essentially implement a wired OR, either by relying on an ordered FAT-tree topology in the off-chip domain, such as in IBM Blue Gene/L [28], or 1-to-M connectivity via on-chip global broadcast wires [5, 54], or all-to-all connectivity between special-purpose registers among a cluster of nodes [80, 18]. These works also add tables to track barriers, thus placing a limit on the number of active barriers at any point in time and adding area/power overheads. Our FANIN is much more general purpose because we target an unordered, distributed on-chip network with any kind of M-to-1 control flow (acknowledgements, tokens, barriers), without adding dedicated 1-to-M wires or extra storage structures.

5.3 Walk-through Example

We will use the term "M:1" to refer to the communication flow in which M cores generate one response message each for the same destination core and memory address, acknowledging the preceding multicast. For convenience, we will use the term "ACK" for each individual flit in the M:1 flow, though in principle it is not restricted to just

coherence	network	**ack_count**	**ack_id**		**is_ack**	valid
(type, addr, etc.)	(type, route, VCid)		dest_id	m_id		
<40>	<9>	<6>	<6>	<5>	<1>	<1>

Figure 5-1: ACK flit format.

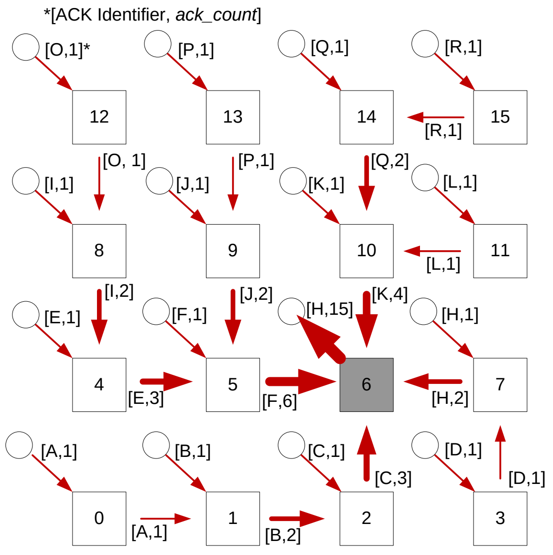


Figure 5-2: ACK Aggregation Example.

acknowledgements and can work for tokens, barriers, etc. The message format we use for ACKs (assuming a 64-core system) is shown in Figure 5-1. A single-bit *is_ack* is set to identify an ACK. We assume that each ACK flit carries a counter called *ack_count* holding the number of ACKs it represents. This field is log_2(N)-bits wide, where N is the maximum number of ACKs that could be received[1]. All ACKs within the same M:1 flow carry the same *ack_id* which will be explained later in Section 5.5.2.

ACKs for an M:1 flow are aggregated/reduced at network routers as they move towards their common destination. To aggregate two ACKs that are part of the same M:1 flow, one of the ACK flits is dropped, and its *ack_count* added to the *ack_count* of the other flit.

Figure 5-2 presents a walk-through example for 16 cores. All NICs (except NIC 6) are sending an ACK to NIC 6. We arbitrarily label these ACKs from A to O for illustration purposes; the individual identity of ACKs is not important, only the *ack_count* they carry is. ACK O is injected from NIC 12 with an *ack_count* of 1. It is

[1]Response flits in certain protocols like Token Coherence [61] already carry such a counter, because cores can respond with multiple tokens at a time.

aggregated into ACK I at Router 8 and dropped; ACK I is aggregated into E, and E is aggregated into F. At Router 6, ACKs F, K, H and C - from West, North, East and South input ports respectively - are aggregated into ACK H. H is sent up to the NIC with an *ack_count* of 15, instead of the NIC having to wait for individual ACKs from all the 15 senders.

Realizing a perfect aggregation of 15 flits into 1 is non-trivial. The reason is that ACKs are generated by the different cores at *different* times and take a *different* number of cycles to reach the next routers, during which time other ACKs for the same M:1 flow might have already left that router. It is also important to note that perfect aggregation is not necessary for correctness.

The rest of the chapter describes how FANIN enables in-network aggregation, and tries to approach the performance of a perfect M-to-1 aggregation network.

5.4 *rWhirl*: Synchronized Routing

We first ensure that all ACKs for a particular M:1 flow follow a synchronized route. This enables ACKs at intermediate routers to know (1) which ports they need to poll for other ACKs, and (2) when all possible aggregations at that router complete. For instance, in Figure 5-2, the ACK at the injection port of Router 5 only needs to poll input ports West and North; once it merges ACKs from both directions, it can move ahead. It does not need to wait for an ACK from the South port because ACK B from Router 1 will reach the destination via Router 2, and not Router 5, in this particular routing policy.

While a fixed route (like XY) by all ACKs serves this purpose, it would result in heavy congestion across the Y links leading to the destination because the destination would become a hot-spot node. Instead, we make all ACKs for an M:1 flow follow the *reverse* path of the 1-to-M *Whirl* route they were on, as shown in Figure 5-3. We term this as *reverse Whirl* or *rWhirl*. This is realized by embedding the received *Whirl* route (4-bit $LTB_W, LTB_N, LTB_E, LTB_S$) from the broadcast into the response flit. Each router on the response path can now decode these bits to compute the 'LTB, RTB'

for each direction for the original broadcast and estimate the output port for the ACK, as shown in Figure 5-3[2].

Deadlock avoidance. *rWhirl* allows all possible turns, and thus requires a deadlock avoidance mechanism. We avoid deadlocks by using the same VC partitioning technique as we did for *Whirl* (see Figure 4-4). All ACKs that start going South are forced to use VC-a, and not VC-b, until they turn, after which they can use any VC. ACKs going in other directions can use any VC. This implements a deadlock-free South-last turn model within VC-b, and guarantees deadlock freedom.

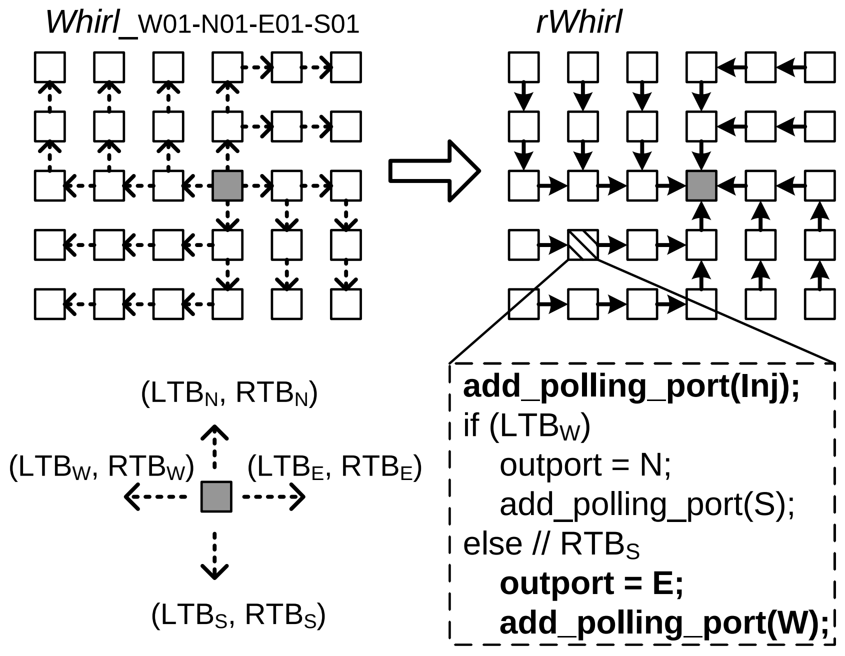
5.5 FANIN Flow Control/Protocol

5.5.1 master ACKs

Who does the aggregation? The first ACK that arrives at a router is responsible for aggregating ACKs for that M:1 flow at that router, and we call it the *master* ACK. In most cases, the first ACK to arrive at a router will be at the injection port. This is because any multicast that went through this router would have delivered the multicast flit to the local NIC earlier than delivering it to the neighbor's NIC, and consequently the response ACKs would follow that order. Exceptions to this could occur due to congestion at cache controllers and cores. This master ACK gets buffered on arrival. In parallel, it determines which input ports it needs to poll based on the *rWhirl* route and the router's location relative to the destination. Figure 5-3 shows an example of the polling logic.

How is the aggregation done? The master ACK flit checks the incoming links at its polling ports every cycle. It does not poll flits already buffered in the router, which will be explained later. Whenever a new ACK arrives at the router (indicated by the *is_ack*-bit) at an input port, its *ack_id* (see Section 5.5.2 for details) is compared against all master ACKs (from different M:1 flows) polling this input port. On a

[2]In certain protocols such as HyperTransport [9], the 1:M source and the M:1 receiver are not the same. This is not a problem because *rWhirl* essentially determines a load-balanced and synchronized route for the ACKs. They do not have to follow the exact reverse route of the broadcast.



```
add_polling_port(Inj);
if (LTB_W)
    outport = N;
    add_polling_port(S);
else // RTB_S
    outport = E;
    add_polling_port(W);
```

Figure 5-3: rWhirl with sample pseudo-code.

match, the master ACK updates its *ack_count*, and the flit that arrived is simply dropped. Dropping the flit entails sending a credit back to the upstream router for the VC it was going to be buffered in.

What happens to polling ports after aggregation? Once the ACK flit aggregates a flit from an input port, it does not remove that port from its polling list to account for inefficient aggregation at upstream routers. For instance, in Figure 5-2, if Router 8 failed to aggregate ACK O, both ACKs O and I would arrive at Router 4 from the North port, so Router 4 should continue polling this port.

Can there be multiple masters for the same M:1 flow at a router? If an ACK became a master, it means there is no other ACK for the same M:1 flow at this router, else there would have been an earlier master that would have aggregated and dropped it on arrival. Thus, master ACKs do not need to poll buffered flits within the router. There is, however, the corner case of two ACKs arriving in the same cycle from different input ports, in which case they would both become master ACKs if no other master ACK was looking out for them. To handle this special case, we (a) make each input port store the VCid of the ACK flit that arrived in the previous

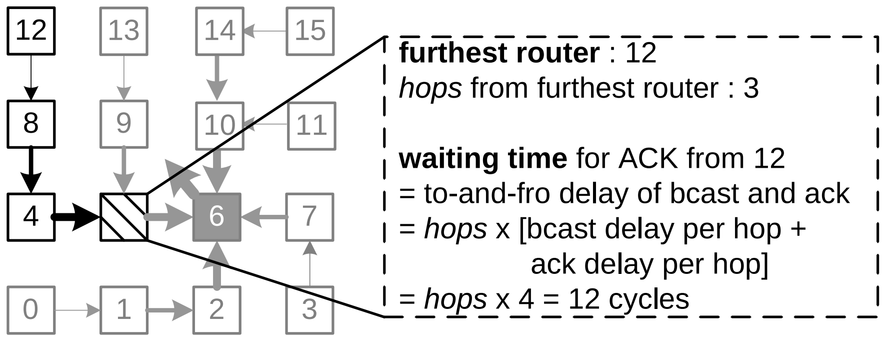

12
13
14
15
8
9
10
11
4
6
7
0
1
2
3
furthest router : 12
hops from furthest router : 3
waiting time for ACK from 12
= to-and-fro delay of bcast and ack
= hops x [bcast delay per hop + ack delay per hop]
= hops x 4 = 12 cycles

Figure 5-4: Waiting heuristic for ACKs.

cycle, and (b) give ports an arbitrary static priority: $Inj > W > N > E > S$. In the first cycle after getting buffered, the master ACKs check the last arrival VCs at their polling ports (in addition to polling the links). If they find a match, the master ACK with higher priority aggregates the ACKs with lower priorities, and the latter VCs at the respective ports are made free.

Can we achieve perfect aggregation? Perfect aggregation, i.e., aggregating all ACKs part of a M:1 flow into one ACK, is not necessary for correctness; moreover, it cannot be achieved if ACK flits keep winning switch allocation at intermediate routers and proceed to their destination. In principle, a master ACK from the NIC could wait indefinitely at a router till it aggregates all the possible ACKs for that flow which will pass through that router, to realize the aggregation presented earlier in Figure 5-2. However, this can cause deadlocks. For instance, a master ACK $F1$ at Router 5 might be waiting for ACK $E1$ from Router 4, while a master ACK $E2$ at Router 4 might be waiting for ACK $F2$ from Router 5, both blocking the injection port buffers at their respective routers, preventing the injection of $E1$ and $F2$ respectively. This can occur because controllers could inject ACKs for two *different* M:1 flows in any order as they could have received the broadcast requests in different orders depending on their distance from the source. To avoid deadlocks, we propose a heuristic for a fixed waiting time, during which the master ACK should not arbitrate for the switch and instead only aggregate ACKs; after the waiting time is over, it can start arbitration for the switch while continuing to poll input ports for more potential aggregations.

How long should master ACKs wait at a router? Here, we describe a solution for scenarios in which every node (except the requester) responds with an ACK [9, 1]. For cases when this is not true, such as multicasts with few destinations or certain coherence protocols [61, 58], an opportunistic aggregation by master ACKs, with no explicit waiting is a better alternative (i.e., master ACKs aggregate ACKs in parallel to arbitration for the switch).

The time of arrival of a particular ACK at an intermediate router depends on the location of this router on the mesh, relative to the source of the ACK. The master ACK at a router should wait to aggregate the ACK from the router furthest from it, before it proceeds. Figure 5-4 shows the heuristic we use to compute this waiting time: the time for the ACK from Router 12 to arrive at Router 5 will be greater than or equal to the zero-load, to-and-fro delay of the preceding broadcast, and the current ACK (i.e., $hops \times (2 + 2)$), assuming two cycles at every hop[3] for the broadcast and ACK flits.

The ACK from the local NIC should be the master ACK for the waiting time heuristic to hold. If an ACK arrives from some other port and becomes the master ACK, this means the NIC ACK has not yet arrived or has already left after waiting. This breaks the heuristic's assumptions, so there is no point for this new ACK to wait. The master ACK at the injection port, on arrival, computes $furthest\_hops$, the number of hops between the current router and the furthest router reachable via its polling ports via minimal hops (which is Router 12 in Figure 5-4). The furthest router would be at the corner, and is thus easy to compute using the current coordinates and polling ports. It then waits for $furthest\_hops \times 4$ before starting switch allocation. The master ACKs at all other ports do not wait, and instead perform opportunistic aggregation by polling input links every cycle until they get to use the switch.

The policy of making ACKs wait at routers, while increasing the chances of aggregating other ACKs (thereby reducing traffic), offers a trade-off because waiting ACKs occupy router buffers, throttling new flits from entering the router. Moreover, inefficient waiting could lead to a delayed completion of the preceding request. We evaluate our heuristic in Section 5.7.3.

[3]See Section 4.5.2 and Section 5.6.

5.5.2 Comparison Logic for Aggregation

Two ACKs belonging to the same M:1 flow are identified by identical destinations (6 bits in 8×8 mesh) and memory addresses (32-40 bits). Comparing 38-46 bits at each router at multiple ports is an overkill in terms of area and power, and not very scalable. Hashing the address to fewer bits adds the risk of conflicts during aggregation.

We solve this by leveraging the fact that at any point in time, the number of unique M:1 flows is limited by the number of outstanding multicasts, which in turn is limited by the size of the MSHR at each multicasting cache/directory controller. In our design, each multicasting controller[4] maintains a pool of multicast ids called *m_id*s. Every time a new multicast is sent, it is assigned a unique *m_id* and the controller marks this *m_id* as busy. The number of busy *m_id*s at the controller represents the number of multicast requests for which responses have not yet been received.

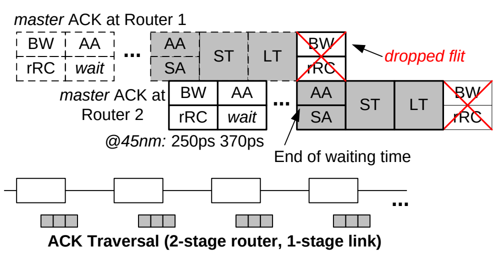
On receiving the multicast, the responding controllers embed the same *m_id* in the ACKs. This ensures that all ACKs belonging to an M:1 communication will have the same $ack\_id = [dest\_id, m\_id]$ and thus this field can be compared for aggregation instead of addresses.

When the multicasting controller receives all expected ACKs, or an unblock[5], it frees the corresponding *m_id*, and can re-issue it to future multicasts. Thus at any point in time the *ack_id* is unique to a particular M-to-1 communication flow.

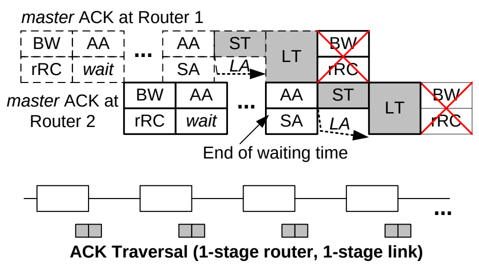
The maximum number of unique *m_id*s required at each multicasting controller is equal to the number of MSHR entries at the controller. In a 64-core CMP, the *dest_id* is 6 bits, while the typical number of MSHR entries is less than 32 [9, 1], giving an *m_id* of 5 bits. This results in an *ack_id* of 11 bits. We can also choose to have fewer *m_id*s than the number of MSHR entries (to reduce the *ack_id* bits further). In this case, if all *m_id*s are busy, and the multicasting controller needs to send out a new

[4]This could be the requester [61, 1] or the home node [9, 58].

[5]In protocols like HyperTransport [9], all ACKs go to the requester, which then sends an unblock message to its home node, which is the multicasting controller.



(a) **Regular FANIN pipeline.**



(b) **Optimized FANIN pipeline.**

Figure 5-5: FANIN pipeline optimizations to realize single-cycle aggregation router. *The critical cycles (adding to overall traversal delay) of the pipelines are shaded in gray. The stages are: BW: buffer write, rRC:* rWhirl *route computation, AA: ACK aggregation, SA: switch allocation, ST: switch traversal, LT: link traversal, LA: lookahead.*

multicast, it assigns it an *m_id* of -1. ACKs with *m_id* of -1 are not aggregated in the network, thus maintaining correctness[6].

5.6 Single-cycle FANIN Router

FANIN's regular pipeline is shown in Figure 5-5(a), along with the synthesized critical path delays for the *rWhirl* (rRC) and ACK aggregation (AA) steps. We discuss Router 2 without any loss of generality. We define critical stages as the pipeline stages that add to the number of cycles per hop for the aggregated ACKs. The master ACK at Router 2 always needs to get buffered because it performs the aggregation and needs to wait for incoming ACKs. However, this part of the pipeline is overlapped by the flit arrival from Router 1 for which this master ACK is waiting, and is thus non-critical. Once the flit from Router 1 arrives, it gets aggregated and dropped, and the master ACK at Router 2 starts switch allocation (SA). It then performs switch traversal (ST). These two router pipeline stages are critical. We leverage bypassing, sending an lookahead (LA) one-cycle ahead of the regular flit to shrink the number of critical stages in the router to one, as shown in Figure 5-5(b). The lookahead in this scenario needs to carry the 11-bit *ack_id*, 1-bit *ack_bit*, and 6-bit *ack_count*. This

[6]GETS requests in Token Coherence [61] are also assigned an *m_id* of -1 since MSHR entries can become free before all tokens are received in some cases.

lookahead can perform the aggregation, while the actual flit traverses the link (and is then dropped the next cycle on aggregation). This optimized ACK traversal is shown in Figure 5-5(b) and presents two critical stages at every hop for ACKs (one cycle in router, one in link). This FANIN pipeline is used for the response message class of the network. The request message class will follow the FANOUT pipeline described earlier in Section 4.5.2.

Ideal M-to-1 Traversal. In summary, FANIN enables ACKs from the four quadrants of the chip to reach the destination router in a synchronized manner (using *rWhirl*) as four aggregated ACKs that incurred 1-cycle router and 1-cycle link critical delays at all intermediate hops (using intelligent waiting and lookaheads). At the destination, these ACKs get opportunistically aggregated into one ACK and proceed to the NIC, thus reducing network load from M to a single ACK and pushing energy-delay-throughput to the ideal.

5.7 Evaluations

We present full-system evaluations of FANIN with HyperTransport and Token Coherence. Table 5.1 and Table 5.2 present our CPU/Memory and NoC parameters. We use the FANOUT NoC presented in Chapter 4 as our baseline design. As before, our IDEAL (t_r=1) is a NoC with 1-cycle routers and no contention[7].

Table 5.1: CPU and Memory Parameters.

Processors	64 in-order SPARC
L1 Caches	Private 32 kB I&D
L2 Caches	Private 1MB per core
Coherence Protocol	(1) HyperTransport [9] (HT) (2) Token Coherence [61] (TC)
DRAM Latency	70ns

Table 5.2: Network Parameters.

Topology	8×8 mesh
Router Ports	5
Ctrl VCs/port	12, 1-flit deep
Data VCs/port	3, 3-flit deep
Flit size	128 bits
Link length	1mm

[7]Section 2.3.1 describes the implementation of the contention-less ideal NoC.

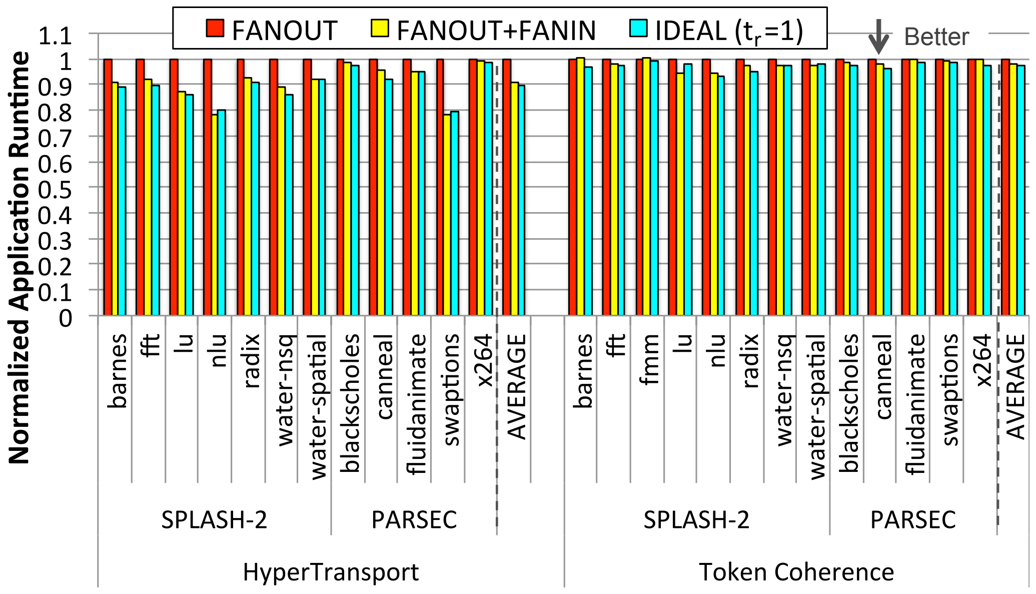


Figure 5-6: Full-system Application Runtime.

5.7.1 Application Runtime

Figure 5-6 shows the normalized full-system application runtime with FANOUT+FANIN for both protocols. With FANOUT, we saw a runtime improvement of 10% for HT in Figure 4-14, which was 9.5% off the ideal. FANIN bridges this gap. FANIN reduces runtime by a further 9.2% over FANOUT. For TC, FANIN lowers runtime by a 2% over FANOUT. These results reiterate that the progress of applications is limited more by ACKs in HT, and by broadcasts in TC. Compared to IDEAL (t_r=1), FANOUT+FANIN is off by less than 1% for both HT and TC.

5.7.2 Network Latency

The full-system runtime behavior seen earlier can be understood by looking at the network latency impact in Figure 5-7. FANOUT+FANIN allows HT to achieve an average on-chip network latency of 22.6 cycles, which is just 0.5-cycle more latency per hop on average than the ideal 1-cycle-router data-path. There is no significant latency reduction for TC than that already provided by FANOUT since FANIN only affects 2% of the flows in this protocol.

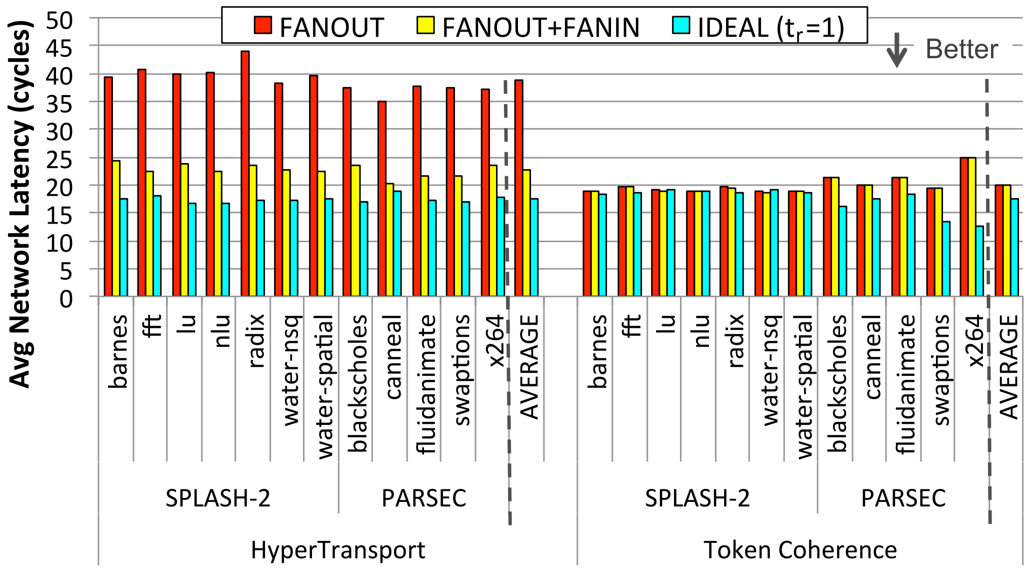


Figure 5-7: Average Network Latency.

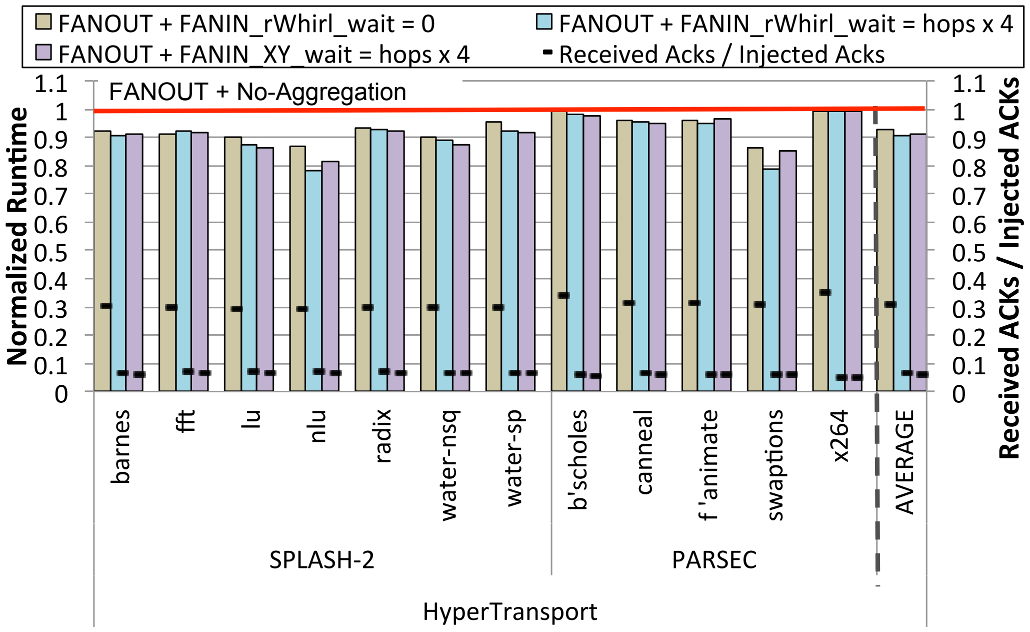


Figure 5-8: Impact of routing and flow control components.

5.7.3 Impact of components of FANIN.

We discuss the impact of FANIN on HT in Figure 5-8 which plots full-system runtime. We first evaluate the performance of FANIN with opportunistic aggregation[8]. This results in a 6.9% runtime improvement. The ratio of received-to-injected ACKs

[8]The master ACKs do not explicitly wait; instead they poll input ports for aggregation opportunities in parallel to switch arbitration.

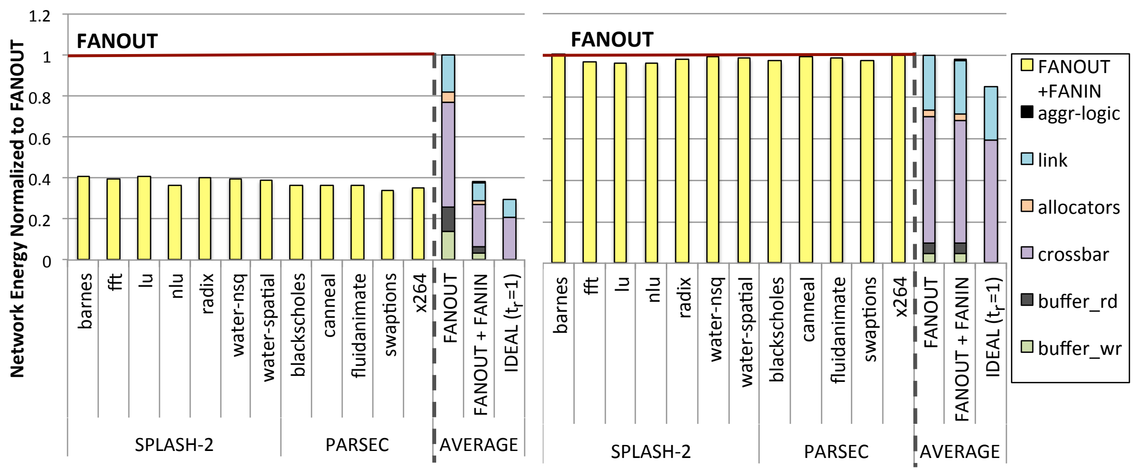
goes down to 0.31 on average. The intuition for two-thirds of the ACKs for M:1 flows getting aggregated, even with opportunistic aggregation, is that during the M:1 burst there is heavy congestion at routers leading up to the hot-spot destination. Though this increases the arbitration cycles, it ends up increasing the opportunity for aggregation as well, leading to an an overall reduction in traffic and runtime.

Next, we add our heuristic of waiting for $furthest\_hops \times 4$ cycles. The ideal ratio of received-to injected ACKs should be 1/63 (all cores except the requester send an ACK) = 0.015. However, because we do not perform any waiting at the destination router, the destination NIC should receive four ACKs, instead of 63, making the best achievable ratio = 4/64 = 0.0625. We observe that the received-to-injected ACKs goes down from 0.31 to 0.065 with our heuristic, which is only 4% higher than the best FANIN can achieve. The runtime goes down by another 3%. While the waiting heuristic is needed to approach the ideal network's runtime, higher wait times degraded runtime because waiting ACKs reduce available router buffers, and inefficient waiting could delay request completion.

We also study the impact of *rWhirl* versus XY routing. Interestingly, XY performs comparably to, and sometimes slightly better than, *rWhirl* for most benchmarks, except *nlu* and *swaptions*. The reason is that XY forces all ACKs to travel X first, which results in all routers in the North/South of the destination receiving ACKs from three directions (while routers in East/West of the destination receive ACKs from only one direction). If the wait-time is perfect, this should not matter. However, because it is only a heuristic, the waiting master ACK at the North/South routers in XY ends up performing 3% more aggregations than in the *rWhirl* case. This result highlights that, for ACKs, the waiting and aggregation ratio has a higher impact on runtime than the load-balancing across network paths.

5.7.4 Network Energy.

Figure 5-9 plots the breakdown of network energy for FANOUT+FANIN for HT and TC, normalized to FANOUT. The energy consumed by the AA logic (comparators and adders) is also accounted for. For HT, the network energy does down by 62% on



(a) **HyperTransport.** (b) **Token Coherence.**

Figure 5-9: Network Energy.

average. This is in part because 93.5% aggregation of ACKs are aggregated, leading to reduction of traffic and hence fewer buffer, crossbar and link traversals. In addition, lower NoC contention due to FANIN allows more buffer bypasses for other kinds of traffic as well. For TC, energy goes down by 2.4%. Compared to the IDEAL (t_r=1) in energy[9], FANOUT+FANIN is off by 9.6% on average for both HT and TC, as opposed to 32.1% in the BASE_fork@rtr design described in Chapter 4. If we reduce the number of VCs from 12 for req vnet and 3 for resp vnet (Table 5.2) - determined from turnaround time of BASE_fork@rtr - to 8 and 2 respectively, FANOUT+FANIN is still less than 1% off IDEAL (t_r=1) runtime but consumes 17.7% lower energy, which is 7.9% off IDEAL (t_r=1) energy.

5.8 Chapter Summary

In this chapter, we presented FANIN, an in-network aggregation methodology for M-to-1 flows, which occur in a class of shared memory coherence protocols. We also added optimizations for synchronized routing, intelligent waiting to maximize aggregation, and single-cycle router pipelines. FANIN, together with FANOUT which was presented in the previous chapter, can provide scalability to shared memory

[9]The energy consumed in the ideal networks is just data-path energy, i.e., ST and LT.

coherence protocols that frequently use broadcasts and wide multicasts. The ideas in FANIN can also be ported to enhance user-level messaging systems that use M-to-1 communication.

So far in this thesis, we have successfully demonstrated single-cycle per-hop NoC traversals for 1-to-1, 1-to-Many and Many-to-1 traffic flows. In the next chapter, we push further towards our goal of realizing dedicated wire connections in a shared NoC by enabling multi-hop traversals to occur within a single-cycle.

6 Single-cycle Multi-hop NoC for 1-to-1, 1-to-Many and Many-to-1 Traffic

"Begin at the beginning," the King said very gravely,
"and go on till you come to the end: then stop."
- Lewis Carrol, *Alice in Wonderland*

This chapter presents a NoC architecture that allows flits to dynamically create and traverse multi-hop routes, potentially all the way from the source to the destination, within a single-cycle. This design aims to remove the fundamental dependence of the network latency on the number of hops H in Equation 1.1. The key enabler for this NoC architecture is single-cycle multi-hop repeated wires, which are presented in Appendix A.

6.1 Introduction

In Chapter 2, we presented the following fundamental equation for the latency of a packet in the NoC:

$$T_P = H \cdot (t_r + t_w) + T_s + \sum_{h=1}^{H} t_c(h) \tag{6.1}$$

It has a fixed component for router (t_r) + link (t_w) delay, which gets multiplied by the number of hops H; a constant serialization delay T_s for multi-flit packets equal to number of flits minus one, i.e., $(\lceil L/b \rceil - 1)$, where L is the packet length and b is the

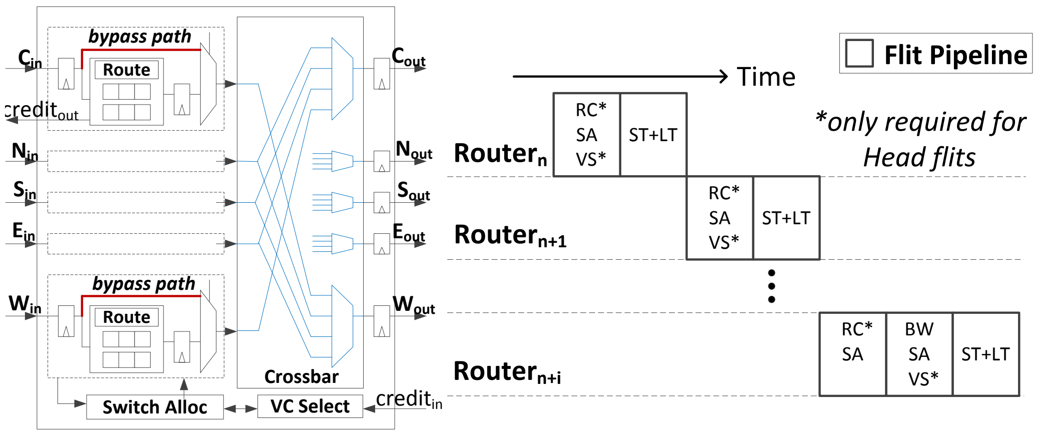


Figure 6-1: BASELINE (t_r=1) Router Microarchitecture and Pipeline.

link bandwidth; and a variable delay depending on contention at every hop ($t_c(h)$).

So far in this thesis, we have managed to reduce the fixed component to its minimum possible value: a single-cycle in the router and a single-cycle in the link connecting adjacent routers. The microarchitecture and pipeline of such a BASELINE (t_r=1) router is shown in Figure 6-1[1]. However, the latency still goes up linearly with H. As core counts increase, H inevitably increases (linearly with k in a $k \times k$ mesh). As we design 1024 core chips [3, 58, 33, 42] for the exascale era, high hop counts will lead to horrendous on-chip network traversal latency and energy creating a stumbling block to core count scaling.

One proposed approach for reducing network latency is to modify the mesh topology and add extra dedicated links between certain physically distant routers [32, 44, 13, 48, 74]. This reduces H and thus reduces the number of routers on the route, lowering latency. These long point-to-point links can be engineered to incur a delay of only a single-cycle by using repeated wires[2] or equalized low-swing wires [45]. However, these solutions are far from perfect. (1) High-radix routers have a high number of input and output ports, leading to increased complexity of the routing, allocation and crossbar blocks, increasing router delay t_r and router power. (2) If each extra link is the same width as the ones in the underlying mesh, the buffer + crossbar area and

[1]For simplicity, we merge the *lookahead* and flit pipelines which were shown separately in Chapter 3. The actions within the router include allocations and routing (performed by lookaheads) and flit buffering (if allocation fails). This is followed by the switch+link traversal of the lookaheads/flits.

[2]In Appendix A we show that repeated wires can transmit signals across 13+ mm within a GHz.

power increase dramatically. Instead, thinner channels (i.e., smaller b) are often used to remove the area or power penalty, but this has a performance penalty in terms of higher serialization delay T_s. (3) The explicit links can help reduce latency only for traffic that maps well to this new topology.

We propose an alternate approach to realize a single-cycle network traversal. We embed repeated links within the conventional data-path of mesh routers, and present a flow control technique that allows flits to create virtual multi-hop paths which can be traversed within a single-cycle. We optimize network latency as follows:

$$T_P = \lceil (H/HPC) \rceil \cdot (t_r + t_w) + T_s + \sum_{h=1}^{H} t_c(h) \tag{6.2}$$

where *HPC* stands for number of Hops Per Cycle. We reduce the effective number of hops to $\lceil (H/HPC) \rceil$, without adding any additional physical wires in the data-path or reducing b like the high-radix router solutions do.

Our proposed NoC is named SMART, for Single-cycle Multi-hop Asynchronous Repeated Traversal [50]. On a 64-core mesh, synthetic unicast traffic shows 5-8X reduction in average network latency; full-system SPLASH-2 and PARSEC traffic shows 27/52% reduction in average runtime for Private/Shared L2 in a full-state directory protocol, compared to a BASELINE (t_r=1)[3] design. SMART, enhanced with multicast support, speeds up broadcast delivery by 84% and provides 19% runtime reduction in the broadcast-intensive Token Coherence protocol. With HyperTransport, which has a mix of unicast, broadcast and reduction/acknowledgement traffic, SMART offers a 15% runtime improvement.

The rest of the chapter is organized as follows. Section 6.2 introduces the SMART interconnect, and how it is embedded into a router. Section 6.3 demonstrates the design for a k-ary 1-Mesh, and Section 6.4 extends it to a k-ary 2-Mesh. Section 6.5 and 6.6 enhances the SMART NoC to support 1-to-Many and Many-to-1 traffic flows respectively. Section 6.7 presents implementation details. Section 6.8 presents our evaluations. Section 6.9 contrasts against prior art and Section 6.10 concludes.

[3] A single-cycle router at every hop.

6.2 The SMART Interconnect

Router logic delay limits the network frequency to 1-2GHz at 45nm [38, 66]. Link drivers are accordingly sized to drive a signal up to 1-hop (1mm in this work) in 0.5-1ns, before it is latched at the next router. But wires can transmit signals much further within this target frequency. Appendix A performs a detailed design-space exploration of repeated wire[4] delay as a function of different design parameters. We observe that repeated wires can go 13-19mm within a ns at $3\times DRC_{min}$ wire spacing, and 1mm repeater spacing. SMART exploits the positive slack in the link traversal stage by replacing clocked link drivers by asynchronous repeaters at every hop, and removing the constraint of latching signals at every router. This allows signals to be transmitted up to multiple hops within a cycle before they are latched at the destination router. The maximum number of hops that can be traversed within a cycle, or HPC_{max}, is an architectural parameter that can be inferred from the maximum length that can be traversed by a signal on a repeated wire at a given technology; this is shown in Equation 6.3.

$$HPC_{max} = \frac{(max\ mm\ per\ ns) \times (clock\ period\ in\ ns)}{(tile\ width\ in\ mm)} \tag{6.3}$$

For a 1mm tile width and a 1ns clock period that we assume for our experiments, we get a HPC_{max} of 13-19 for a repeated wire at 45nm. If we choose a 2mm tile width, or a 2GHz frequency, HPC_{max} will go down by half. Clock-less/asynchronous repeaters also consume 14.3% lower energy/bit/mm than conventional clocked drivers, giving us a win-win.

SMART is a better solution for exploiting the slack than deeper pipelining of the router with a higher clock frequency (e.g. Intel's 80-core 5GHz 5-stage router [37]) which, even if it were possible to do, does not reduce traversal latency (only improves throughput), and adds huge power overheads due to pipeline registers.

Figure 6-2 shows a 5-ported SMART router for a mesh network. For simplicity, we

[4]Adding asynchronous repeaters, i.e., an inverter or a pair of inverters, is a standard way of reducing wire delay [45, 70].

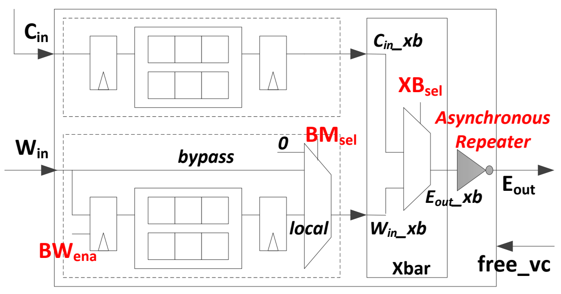
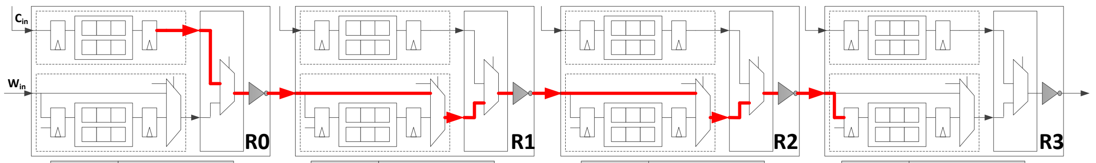


Figure 6-2: SMART Router Microarchitecture.



	R0	R1	R2	R3
BW_{ena}	0	0	0	1
BM_{sel}	0	bypass	bypass	0
XB_{sel}	C_{in}->E_{out}	W_{in}->E_{out}	W_{in}->E_{out}	X

Figure 6-3: Traversal over a SMART path.

only show $Core_{in}(C_{in})$[5], $West_{in}(W_{in})$ and $East_{out}(E_{out})$ ports. All other input ports are identical to W_{in}, and all other output ports are identical to E_{out}. The delay at each hop includes not just the repeater and link segment delay (as in a pure repeated wire), but also the delay through the muxes (2:1 bypass and 4:1 Xbar). This reduces HPC_{max} to 11 at 1GHz, as explained later in Section 6.7.

Figure 6-2 shows the three primary components of the design: (a) Buffer Write enable (BW_{ena}) at the input flip flop which determines if the input signal is latched or not, (2) Bypass Mux select (BM_{sel}) at the input of the crossbar to choose between the local buffered flit, and the bypassing flit on the link, and (3) Crossbar select (XB_{sel}). Figure 6-3 shows an example of a multi-hop traversal: a flit from Router R0 traverses 3-hops within a cycle, till it is latched at R3. The crossbars at R1 and R2 are preset to connect W_{in} to E_{out}, with their BM_{sel} preset to choose bypass over local. A SMART path can thus be created by appropriately setting BW_{ena}, BM_{sel}, and XB_{sel} at intermediate routers. In the next section, we describe the flow control to preset these signals.

[5]C_{in} does not have a bypass path like the other ports because all flits from the NIC have to get buffered at the first router, before they can create SMART paths, as explained later in Section 6.3.

Table 6.1: Terminology

Term	Meaning
HPC	Hops Per Cycle. The number of hops traversed in a cycle by any flit.
HPC$_{max}$	Maximum number of hops that can be traversed in a cycle by a flit. This is fixed at design time.
SMART-hop	*Multi-hop* path traversed in a *Single-cycle* via a SMART link. It could be straight, or have turns. The length of a SMART-hop can vary anywhere from 1-hop to HPC_{max}.
injection router	First router on the route. The source NIC injects a flit into the C_{in} port of this router.
ejection router	Last router on the route. This router ejects a flit out of the C_{out} port to the destination NIC.
start router	Router from which any SMART-hop starts. This could be the injection router, or any router along the route.
inter router	Any intermediate router on a SMART-hop.
stop router	Router at which any SMART-hop ends. This could be the ejection router or any router along the route.
turn router	Router at a turn (W_{in}/E_{in} to N_{out}/S_{out}, or N_{in}/S_{in} to W_{out}/E_{out}) along the route.
local flits	Flits buffered at any start router.
bypass flits	Flits which are bypassing inter routers.
SMART-hop Setup Request (SSR)	Length (in hops) for a requested SMART-hop. For example, *SSR*=*H* indicates a request to stop *H*-hops away. Optimization: Extra *ejection*-bit if requested stop router is ejection router.
premature stop	A flit is forced to stop before its requested *SSR* length.
Prio=Local	Local flits have higher priority over bypass flits, i.e., Priority α 1/(hops_from_start_router).
Prio=Bypass	Bypass flits have higher priority over local flits, i.e., Priority α (hops_from_start_router).
SMART_1D	Design where routers along the dimension (both X and Y) can be bypassed. Flits need to stop at the turn router.
SMART_2D	Design where routers along the dimension and one turn can be bypassed.

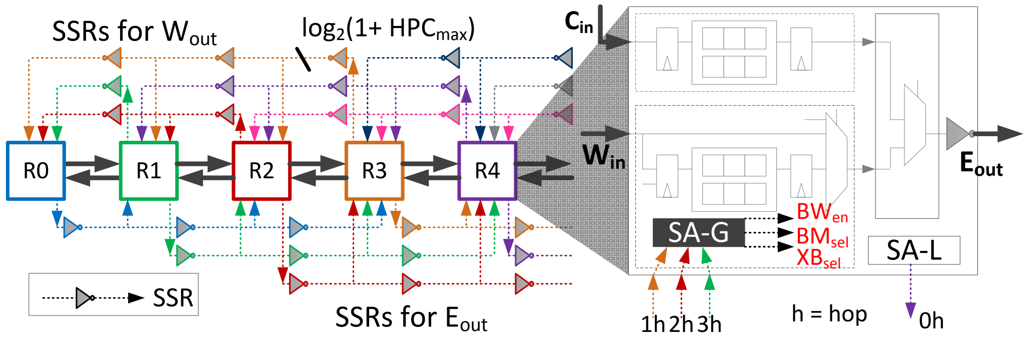


Figure 6-4: k-ary 1-Mesh with dedicated SSR links.

6.3 SMART in a k-ary 1-Mesh

Table 6.1 defines terminology that will be used throughout the chapter. We start by demonstrating how SMART works in a k-ary 1-Mesh, shown in Figure 6-4. Each

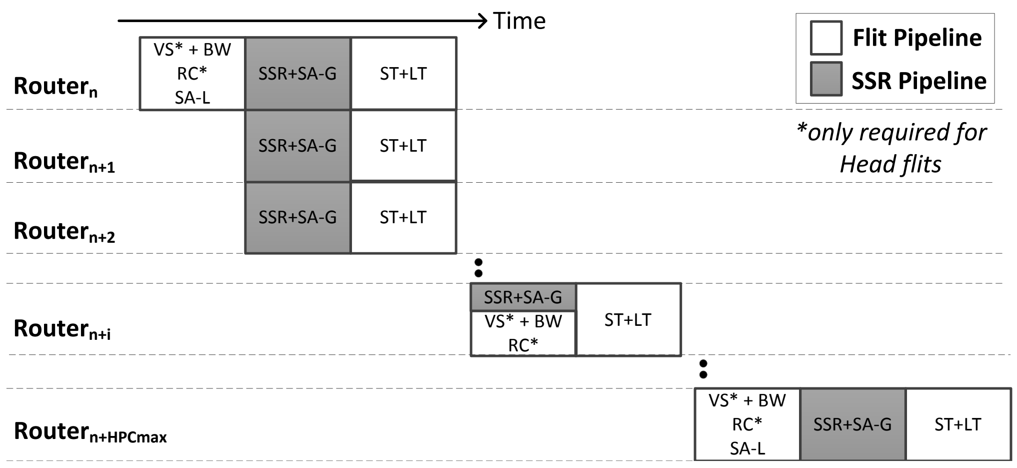


Figure 6-5: SMART Pipeline.

router has 3 ports: West, East and Core[6]. As shown earlier in Figure 6-2, E_{out}_xb can be connected either to C_{in}_xb or W_{in}_xb. W_{in}_xb can be driven either by *bypass*, *local* or 0, depending on BM_{sel}.

The design is called **SMART_1D** (since routers can be bypassed only along one dimension). The design will be extended to a k-ary 2-Mesh to incorporate turns, in Section 6.4. For purposes of illustration, we will assume HPC_{max} to be 3.

6.3.1 SMART-hop Setup Request (SSR)

In the NoCs described so far, buffered flits at every router arbitrate among themselves to gain access to the output ports. We call this Switch Allocation Local (SA-L)[7]. The winner of SA-L (flit or lookahead, depending on implementation) traverses the crossbar and output link to the next router, stops, arbitrates for the next link, and so on. The key challenge in SMART is that flits need to arbitrate for multiple links and the buffer at the end point, all within the same cycle.

The SMART router pipeline is shown in Figure 6-5. A SMART-hop starts from a start router, where flits are buffered. Unlike the baseline router, Switch Allocation in SMART occurs over two cycles: Switch Allocation Local (SA-L) and Switch Allocation Global (SA-G).

[6]For illustration purposes, we only show C_{in}, W_{in} and E_{out} in the figures.
[7]SA-L is identical to the SA stage in the baseline pipeline, described earlier in Section 2.1.4.

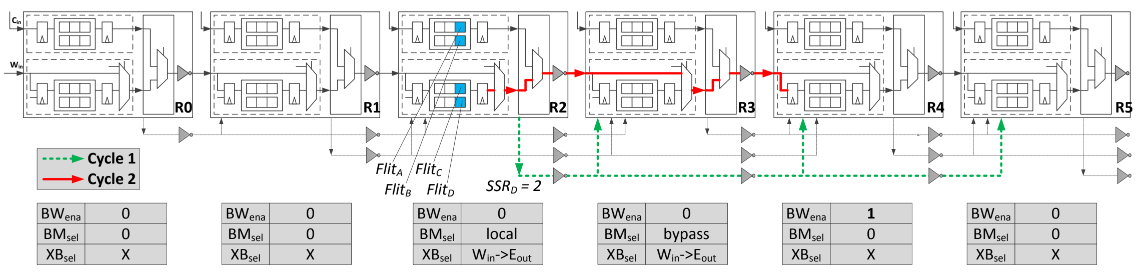
In a SMART NoC, each output port winner from SA-L first broadcasts a SMART-hop setup request (SSR) up to HPC_{max}-hops from that output port. These SSRs - dedicated repeated wires (which are inherently multi-drop[8]) on the control-path that connect every router to a neighborhood of up to the HPC_{max} help preset the intermediate routers for a multi-hop bypass path. They are shown in Figure 6-4. SSRs are $log_2(1+HPC_{max})$-bits wide, and carry the length (in hops) up to which the winning flit wishes to go. For instance, $SSR = 2$ indicates a 2-hop path request. Each flit tries to go as close as possible to its ejection router, hence $SSR = \min(HPC_{max}, H_{remaining})$.

During SA-G, all inter routers arbitrate among the $SSRs$ they receive, to set the BW_{ena}, BM_{sel} and XB_{sel} signals. The arbiters guarantee that only *one* flit will be allowed access to any particular input/output port of the crossbar. In the next cycle (ST+LT), SA-L winners that also won SA-G at their start routers traverse the crossbar and links upto multiple hops till they are stopped by BW_{ena} at some router. Thus flits spend at least 2 cycles (SA-L and SA-G) at a start router before they can use the switch. SSR traversal and SA-G occur serially within the same cycle (see Section 6.7 for timing implications).

We illustrate the traversal mechanism with an example. In Figure 6-6, Router R2 has $Flit_A$ and $Flit_B$ buffered at C_{in}, and $Flit_C$ and $Flit_D$ buffered at W_{in}, all requesting E_{out}. Suppose $Flit_D$ wins SA-L during Cycle-0. In Cycle-1, it sends out $SSR_D = 2$ (i.e., request to stop at R4) out of E_{out} to Routers R3, R4 and R5. SA-G is performed at each router. At R2, which is 0-hops away ($< SSR_D$), BM_{sel} = local, $XB_{sel} = W_{in}\_xb \rightarrow E_{out}\_xb$. At R3, which is 1-hop away ($< SSR_D$), BM_{sel} = bypass, $XB_{sel} = W_{in}\_xb \rightarrow E_{out}\_xb$. At R4, which is 2-hops away ($= SSR_D$), BW_{ena} = high. At R5, which is 3-hops away ($> SSR_D$), SSR_D is ignored[9]. In Cycle-2, $Flit_D$ traverses the crossbars and links at R2 and R3, and is stopped and buffered at R4.

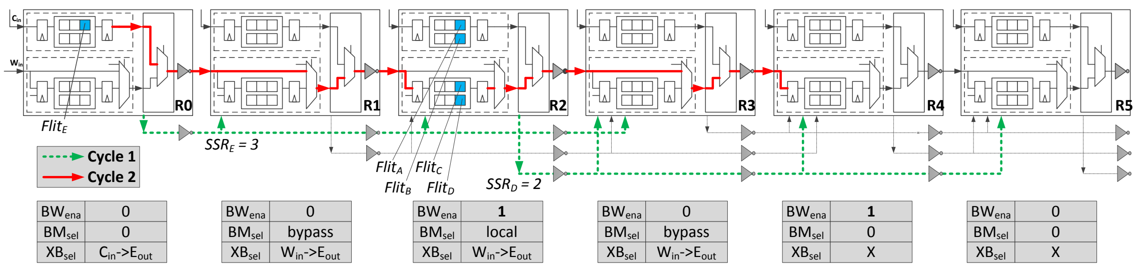
[8]Wire cap is an order of magnitude higher than gate cap, adding no overhead if all nodes connected to the wire receive.

[9]SSRs could be made one-hot to remove the decoder at the receivers. However, this would increase the number of SSR wires linearly with HPC_{max} instead of as a log_2. Moreover, even if the receiver's one-hot bit is low, it still needs additional logic to identify its location relative to the high-bit to decide whether to setup a bypass path or ignore the request.



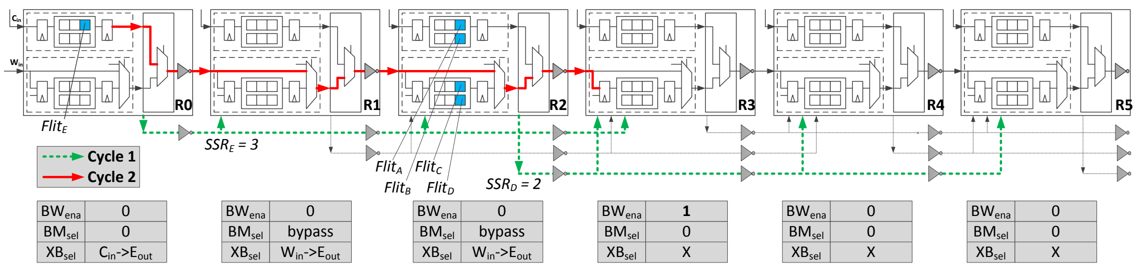
BW_{ena}	0	BW_{ena}	0	BW_{ena}	0	BW_{ena}	0	BW_{ena}	1	BW_{ena}	0
BM_{sel}	0	BM_{sel}	0	BM_{sel}	local	BM_{sel}	bypass	BM_{sel}	0	BM_{sel}	0
XB_{sel}	X	XB_{sel}	X	XB_{sel}	W_{in}->E_{out}	XB_{sel}	W_{in}->E_{out}	XB_{sel}	X	XB_{sel}	X

Figure 6-6: SMART Example: No SSR Conflict.



BW_{ena}	0	BW_{ena}	0	BW_{ena}	1	BW_{ena}	0	BW_{ena}	1	BW_{ena}	0
BM_{sel}	0	BM_{sel}	bypass	BM_{sel}	local	BM_{sel}	bypass	BM_{sel}	0	BM_{sel}	0
XB_{sel}	C_{in}->E_{out}	XB_{sel}	W_{in}->E_{out}	XB_{sel}	W_{in}->E_{out}	XB_{sel}	W_{in}->E_{out}	XB_{sel}	X	XB_{sel}	X

Figure 6-7: SMART Example: SSR Conflict with Prio=Local.



BW_{ena}	0	BW_{ena}	0	BW_{ena}	0	BW_{ena}	1	BW_{ena}	0	BW_{ena}	0
BM_{sel}	0	BM_{sel}	bypass	BM_{sel}	bypass	BM_{sel}	0	BM_{sel}	0	BM_{sel}	0
XB_{sel}	C_{in}->E_{out}	XB_{sel}	W_{in}->E_{out}	XB_{sel}	W_{in}->E_{out}	XB_{sel}	X	XB_{sel}	X	XB_{sel}	X

Figure 6-8: SMART Example: SSR Conflict with Prio=Bypass.

What happens if there are competing *SSR*s? Flits can end up getting prematurely stopped (i.e, before their SSR length) depending on contention between SSRs during SA-G results at the participating routers. In the same example, suppose R0 also wants to send $Flit_E$ 3-hops away to R3, as shown in Figure 6-7. In Cycle-1, R2 sends out SSR_D as before, and in addition R0 sends $SSR_E = 3$ out of E_{out} to R1, R2 and R3. Now at R2 there is a conflict between SSR_D and SSR_E for the W_{in}_xb and E_{out}_xb ports of the crossbar. *SA-G priority* decides which SSR wins the crossbar. More details about priority will be discussed later in Section 6.3.2. For now, let us assume Prio=Local (which is defined in Table 6.1) so $Flit_E$ loses to $Flit_D$. The values of BW_{ena}, BM_{sel} and XB_{sel} at each router for this priority are shown in

Figure 6-7. In Cycle-2, $Flit_E$ traverses the crossbar and link at R0 and R1, but is stopped and buffered at R2. $Flit_D$ traverses the crossbars and links at R2 and R3 and is stopped and buffered at R4. $Flit_E$ now goes through BW and SA-L at R2 before it can send a new SSR and continue its network traversal. A free VC/buffer is guaranteed to exist whenever a flit is made to stop (see Section 6.3.4).

6.3.2 Switch Allocation Global: Priority

Figure 6-8 shows the previous example with Prio=Bypass instead of Prio=Local. This time, in Cycle-2, $Flit_E$ traverses all the way from R0 to R3, while $Flit_D$ is stalled.

Do all routers need to enforce the same priority? Yes. All routers need to collectively agree, in a distributed manner and during the same cycle, on which flit is performing a particular multi-hop traversal. The only information each router has is the SSR from the source; there is no other communication between the routers. Enforcing the same SA-G priority (either Prio=Local or Prio=Bypass) guarantees that the same *relative* priority is maintained between SSRs at each router, ensuring a distributed consensus on which flits get to go and which have to stop. This is required for correctness. In the example discussed earlier in Figures 6-7 and 6-8, BW_{ena} at R3 was low with Prio=Local, and high with Prio=Bypass. Suppose R2 performs Prio=Bypass, but R3 performs Prio=Local, $Flit_E$ will end up going from R0 to R4, instead of stopping at R3. This is not just a misrouting issue, but also a signal integrity issue because HPC_{max} is 3, but the flit was forced to go up to 4 hops in a cycle, and will not be able to reach the clock edge in time. Note that enforcing the same priority is only necessary for SA-G, which corresponds to the global arbitration among SA-L winners at every router. During SA-L, however, different routers/ports can still choose to use different arbiters (round robin, queueing, priority) depending on the desired QoS/ordering mechanism.

Can a flit arrive at a router, even though the router is not expecting it (i.e., false positive[10])? No. All flits that arrive at a router are expected, and

[10]The result of SA-G (BW_{ena}, BM_{sel} and XB_{sel}) at a router is a prediction for the null hypothesis: a flit will arrive the next cycle, and stop/bypass.

will stop/bypass based on the success of their SSR in the previous cycle. This is guaranteed since all routers enforce the same SA-G priority.

Can a flit not arrive at a router, even though the router is expecting it (i.e., false negative)? Yes. It is possible for the router to be setup for stop/bypass for some flit, but no flit arrives. This can happen if that flit is forced to prematurely stop earlier due to some SSR interaction at prior inter routers that the current router is not aware of. For example, suppose a local flit at W_{in} at R1 wants to eject out of C_{out}. A flit from R0 will prematurely stop at R1's W_{in} port if Prio=Local is implemented. However, R2 will still be expecting the flit from R0 to arrive[11]. Unlike false positives, this is not a correctness issue but just a performance (throughput) issue, as we show later in the evaluation Section 6.8.1, since some links go idle which could have potentially been used by other flits if more global information were available.

6.3.3 Ordering

In SMART, any flit can be prematurely stopped based on the interaction of $SSRs$ that cycle. We need to ensure that this does not result in re-ordering between (a) flits of the same packet, or (b) flits from the same source (if point-to-point ordering is required in the coherence protocol).

The first constraint is in routing (relevant to 2D topologies). Multi-flit packets, and point-to-point ordered virtual networks should only use deterministic routes, to ensure that prematurely buffered flits do not end up choosing alternate routes, while bypassing flits continue on the old route.

The second constraint is in SA-G priority. We need to make sure that flits part of a multi-flit packet or flits in a point-to-point ordered vnet do not bypass routers with prematurely buffered flits from that packet or vnet respectively. We add a bit at every input port to track if there is a *prematurely* stopped flit among its buffered flits. When a SSR is received at an input port, and there is either (a) a prematurely buffered Head/Body flit, or (b) a prematurely buffered flit within a point-to-point

[11]The *valid*-bit from the flit is thus used in addition to BW_{ena} when deciding whether to buffer.

ordered vnet, the incoming flit is stopped[12].

6.3.4 Guaranteeing free VC/buffers at stop routers

In a conventional network, a router's output port tracks the IDs of all free VCs at the neighbor's input port. A buffered Head flit chooses a free VCid for its *next* router (neighbor), before it leaves the router. The neighbor signals back when that VCid becomes free. In a SMART network, the challenge is that the *next* router could be any router that can be reached within a cycle. A flit at a start router choosing the VCid before it leaves will not work because (a) it is not guaranteed to reach its presumed next router, and (b) multiple flits at different start routers might end up choosing the same VCid. Instead, we let the VC selection occur at the stop router. Every SMART router receives 1-bit from each neighbor to signal if *at least one* VC is free[13]. During SA-G, if an SSR requests an output port where there is no free VC, BW_{ena} is made high and the corresponding flit is buffered. This solution does not add any extra multi-hop wires for VC signaling. The signaling is still between neighbors. Moreover, it ensures that a Head flit comes into a router's input port only if that input port has free VCs, else the flit is stopped at the previous router.

However, this solution is conservative because a flit will be stopped prematurely if the neighbor's input port does not have free VCs, even if there was no competing SSR at the neighbor and the flit would have bypassed it without having to stop.

How do Body/Tail flits identify which VC to go to at the stop router? Using their *injection_router* id. Every input port maintains a table to map a VCid to an injection router id[14]. Whenever the Head flit is allocated a VC, this table is

[12]While this check is required for Prio=Bypass, at first glance it seems redundant for Prio=Local. This is because Prio=Local inherently guarantees that the buffered flit, say $Flit_{A1}$, will get higher priority than the bypassing flit $Flit_{A2}$, forcing the latter to stop. However, there could be a corner case if we use a separable switch allocator (described earlier in Section 2.1.4) for SA-L. Suppose $Flit_{A1}$ is buffered at W_{in} and requesting for E_{out}. $Flit_{A1}$ loses SA-L's W_{in} arbitration to $Flit_{B1}$. $Flit_{B1}$ loses arbitration for its output port N_{out} to some other $Flit_{C1}$ from some other input port. At the end of SA-L, W_{in} and E_{out} are both open, potentially allowing SA-G to grant $W_{in} \rightarrow E_{out}$ to $Flit_{A2}$. Thus a separate check is required even for Prio=Local.

[13]If the router has multiple virtual networks (vnets) for the coherence protocol, we need a 1-bit free VC signal from the neighbors for each vnet. The SSR also needs to carry the vnet number, so that the inter routers can know which vnet's free VC signal to look at.

[14]The table size equals the number of multi-flit VCs at that input port.

updated. The injection router id entry is cleared when the Tail arrives. The VC is freed when the Tail leaves. We implement private buffers per VC, with depth equal to the maximum number of flits in the packet (i.e., virtual cut-through), to ensure that the Body/Tail will always have a free buffer in its VC[15].

What if two Body/Tail flits with same injection_router id arrive at a router? We guarantee that this will never occur by forcing *all* flits of a packet to leave from an output port of a router, before flits from another packet can leave from that output port (i.e virtual cut-through). This guarantees a unique mapping from injection router id to VCid in the table at every router's input port.

What if a Head bypasses, but Body/Tail is prematurely stopped? The Body/Tail still needs to identify a VCid to get buffered in. To ensure that it does have a VC, we make the Head flit reserve a VC not just at its stop router, but also at all its inter routers, even though it does not stop there. This is done from the *valid*, *type* and *injection_router* fields of the bypassing flit. The Tail flit frees the VCs at all the inter routers. Thus, for multi-flit packets, VCs are reserved at all routers, just like the baseline. But the advantage of SMART is that VCs are reserved and freed at *multiple routers* within the *same cycle*, thus reducing the buffer turnaround time.

6.3.5 Additional Optimizations

We add additional optimizations to SMART to push it towards an IDEAL (T_N=1)[16] described in Chapter 1.

Bypassing the ejection router. So far we have assumed that a flit starting at an injection router traverses one (or more) SMART-hops till the ejection router, where it gets buffered and requests for the C_{out} port. We add an extra *ejection*-bit in the SSR to indicate if the requested stop router corresponds to the ejection router for the packet, and not any intermediate router on the route. If a router receives a SSR from H-hops away with value H (i.e., request to stop there), $H < HPC_{max}$, and

[15]Extending this design to fewer buffers than the number of flits in a packet would involve more signaling, and is left for future work.

[16]Network with 1-cycle latency for all flits.

the *ejection*-bit is high, it arbitrates for C_{out} port during SA-G. If it loses, BW_{ena} is made high.

Bypassing SA-L at low load. We add no-load bypassing [22] to the SMART router. If a flit comes into a router with an empty input port and no SA-L winner for its output port for that cycle, it sends $SSRs$ directly, in parallel to getting buffered, without having to go through SA-L. This reduces t_r at lightly-loaded start routers to 1, instead of 2, as shown in Figure 6-5 for Router$_{n+i}$. Multi-hop traversals within a single-cycle meanwhile happen at all loads.

6.3.6 Summary

In summary, a SMART NoC works as follows:

- Buffered flits at injection/start routers arbitrate locally to choose input/output port winners during SA-L.
- SA-L winners broadcast $SSRs$ along their chosen routes, and each router arbitrates among these $SSRs$ during SA-G.
- SA-G winners traverse multiple crossbars and links asynchronously within a cycle, till they are explicitly stopped and buffered at some router along their route.

In a **SMART_1D** design with both ejection and no-load bypass enabled, if HPC_{max} is larger than the maximum hops in any route, a flit will only spend 2 cycles in the entire network in the best case (1-cycle for SSR and 1-cycle for ST+LT all the way to the destination NIC).

6.4 SMART in a k-ary 2-Mesh

We demonstrate how SMART works in a k-ary 2-Mesh. Each router has 5 ports: West, East, North, South and Core.

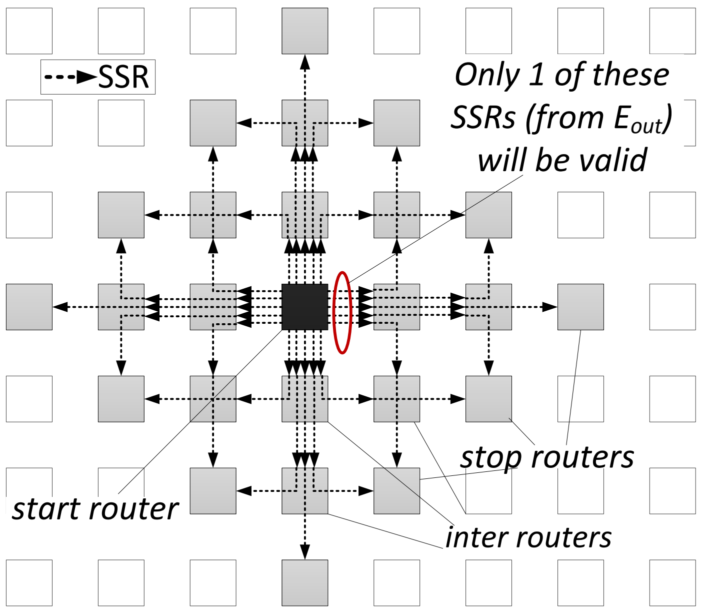


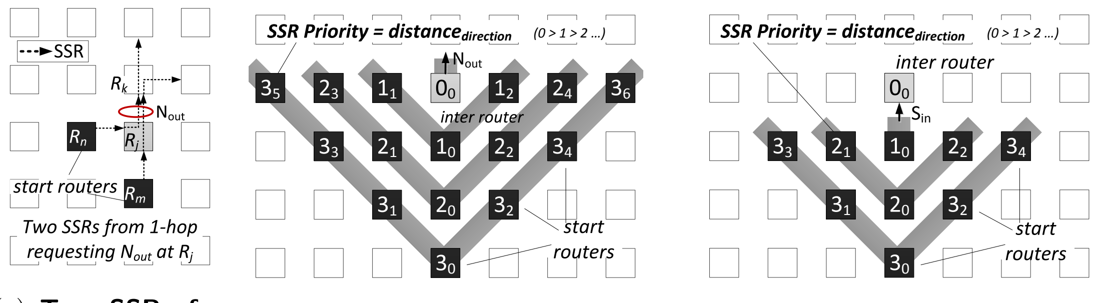
Figure 6-9: k-ary 2-Mesh with SSR wires from shaded start router.

6.4.1 Bypassing routers along dimension

We start with a design where we do not allow bypass at turns, i.e., all flits have to stop at their turn routers. We re-use **SMART_1D** described for a k-ary 1-Mesh in a k-ary 2-Mesh. The extra router ports only increase the complexity of the SA-L stage, since there are multiple local contenders for each output port. Once each router chooses SA-L winners, SA-G remains identical to the description in Section 6.3.1. The E_{out}, W_{out}, N_{out} and S_{out} ports have dedicated SSR wires going out till HPC_{max} along that dimension. Each input port of the router can receive only one SSR from a router that is H-hops away. The SSR requests a stop, or a bypass along that dimension. Flits with turning routes perform their traversal one-dimension at a time, trying to bypass as many routers as possible, and stopping at the turn routers.

6.4.2 Bypassing routers at turns

In a k-ary 2-Mesh topology, all routers within a HPC_{max} neighborhood can be reached within a cycle, as shown in Figure 6-9 by the shaded diamond. We now describe **SMART_2D** which allows flits to bypass both the routers along a dimension and



(a) **Two SSRs for N_{out} port.** (b) **Fixed Priority at N_{out} port of inter router.** (c) **Fixed Priority at S_{in} port of inter router.**

Figure 6-10: SA-G Priority for SMART_2D.

the turn router(s). We add dedicated SSR links for each possible XY/YX path from every router to its HPC_{max} neighbors. Figure 6-9 shows that the E_{out} port has 5 SSR links, in comparison to only one in the SMART_1D design. During the routing stage, the flit chooses one of these possible paths. During the SA-G stage, the router broadcasts *one* SSR out of each output port, on one of these possible paths. We allow only one turn within each HPC_{max} quadrant to simplify the SSR signaling.

SA-G Priority. In the SMART_2D design, there can be more than one SSR from H-hops away, as shown in the example in Figure 6-10(a) for router R_j which receives SSRs from routers R_m and R_n that are both 1-hop away. Router R_k receives the same SSRs. R_j and R_k both need to prioritize the same SSRs to not create false positives[17]. To arbitrate between SSRs from routers that are the same distance away, we add a second level of priority based on *direction*. We arbitrarily choose straight-hops > left-hops > right-hops, where straight, left and right are relative to the input/outport port. Figures 6-10(b) and 6-10(c) plot contours through routers that are the same number of hops away, and highlight each router's relative priority. For the inter router R_j in Figure 6-10(a), the SSR from R_m will have higher priority (1_0) over the one from R_n (1_1) for the N_{out} port, as it is going straight, based on Figure 6-10(b). Similarly at R_k, the SSR from R_m will have higher priority (2_0) over the one from R_n (2_1) for the S_{in} port, based on Figure 6-10(c). Thus both routers R_j and R_k will unambiguously prioritize the flit from R_m to use the links, while the

[17]Section 6.3.2 discussed that false positives can result in misrouted flits or flits trying to bypass beyond HPC_{max}, thus breaking the system. For instance, here if R_j prioritizes the SSR from R_n and R_k prioritizes the SSR from R_m, the flit from R_n will get misrouted.

flit from R_n will stop at Router R_j. We can also infer from Figures 6-10(b) and 6-10(c) that every router sees the same relative priority for SSRs based on distance and direction, thus guaranteeing no false positives.

6.4.3 Routing Choices

Routing choices are orthogonal to the SMART flow control. Flits can choose any deadlock-free route (either dimension-ordered, or enforcing some turn-model [22]), based on any local/global congestion/ordering metric. The only restriction is that only one turn router can be bypassed within HPC_{max} hops, since that is how the dedicated SSR links are laid out. Based on the chosen route, the flit sends an SSR on one of the possible SSR links out of the router's output port.

6.5 SMART for 1-to-Many traffic

In this section, we extend SMART to support 1-to-Many traffic flows, i.e., enable single-cycle chip-wide broadcasts/multicasts. The idea is to enhance the FANOUT router, presented earlier in Section 4, with SMART. We can view a broadcast/multicast as multiple unicasts which wish to use SMART paths.

6.5.1 SMART_1D + FANOUT

We first walk through an example of performing broadcast in a SMART_1D network, also describing the microarchitectural changes as compared to the design presented so far. At any start router, buffered flits first perform multiport SA-L, i.e., mSA described earlier in Section 4.4.3. Multicast flits can place requests for more than one output port - we can use either XY-tree routing or *Whirl* routing (Section 4.3) since SMART is orthogonal to the actual routing path used. A winner is selected independently for each output port (this could be the same flit in case of multicasts). The winner at each output port sends out a SSR along that direction. Each SSR wire has an additional *broadcast bit* to indicate that the flit wishes to perform a broad-

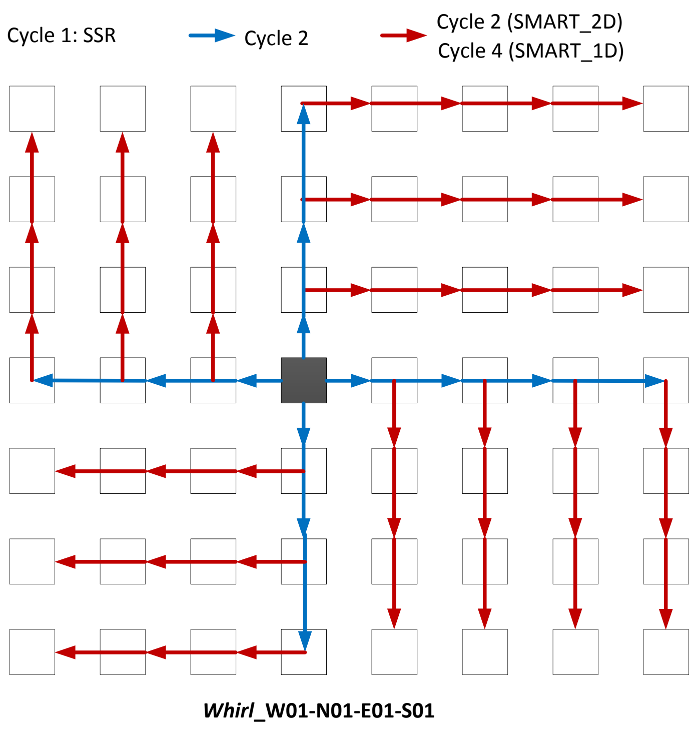


Figure 6-11: Full-chip broadcast with SMART.

cast[18]. At the intermediate routers, SSRs are prioritized as before (i.e., Prio=Local or Prio=Bypass) during SA-G, based on their distance from the router. If the winning SSR has a high broadcast bit, then in addition to the appropriate setup of the BM_{sel} and XB_{sel} signals, BW_{ena} is also made high to retain a copy of the incoming flit at this router. In the next cycle, the flit performs a multi-hop switch and link traversal along the dimension, with a copy being buffered at all intermediate routers.

The buffered copies arbitrate for the C_{out} port and traverse the link to the NIC. In addition, they also create and traverse SMART paths along the other dimension depending on their $Whirl$ LTB and RTB bits. If the buffered flit at a router belongs to a SSR that lost SA-G - which can be inferred from the output of the previous cycle's SA-G - it also needs to continue its traversal along the same dimension to the remaining routers. Multicasts are handled by dropping the copy of the buffered flit if it has been forwarded to the neighbor(s) and the current NIC is not part of the destination set.

[18]We do not need an extra bit if the SSR has unused codes.

6.5.2 SMART_2D + FANOUT

For SMART_2D, the injection router can activate multiple SSR wires in each direction (shown in Figure 6-9), to allow arbitration for multiple output ports at the intermediate routers. During SA-G at the intermediate routers, each output port is allocated independently based on the SSR's 2-level priority (distance and direction), as discussed earlier in Section 6.4.2. At the input port, multiple SSRs from the same source (requesting different output ports) are OR'ed to create one SSR that competes with SSRs from other routers, again based on the fixed 2-level priority. For the input port winner, if more than one output port has been granted, the XB_{sel} is appropriately setup to allow the incoming flit to enter the crossbar and get forked to more than one direction. Since the crossbars in SMART routers have repeaters at every crosspoint (i.e., separate drivers for every output port), forking does not add any extra complexity. A copy of the incoming flit is also retained at every router, similar to the SMART_1D case, for sending to the NIC. This flit also arbitrates for those output ports that the corresponding SSRs lost in the previous cycle.

To reduce overall network latency even further, we add an optimization such that SSRs for broadcasts also arbitrate for the C_{out} port. This allows the incoming flits to potentially fork across multiple directions and fork into the NIC, all within the same cycle, at all routers on the chip; this reduces the minimum possible network delay for broadcasts to be 2-cycles for SMART_2D, and 4-cycles for SMART_1D, as shown in Figure 6-11.

6.5.3 Flow Control

Guaranteeing free VCs at all routers. If a router receives an SSR for a multicast/broadcast flit, but the neighboring router does not have free VCs, the incoming flit is not forwarded along that direction; this is just like the SMART design for unicast traffic. This allows us to avoid the problem of having the flit getting buffered at some nodes, but not all due to VC inavailability. The buffered flit needs to perform a fresh SA-L, followed by SSR and SA-G to continue along that direction.

SA-G Priority. We can implement either of Prio=Local and Prio=Bypass. Since the performance of systems with broadcast traffic is contingent on the time that all nodes receive the flit, Prio=Bypass makes more sense to allow a flow that has started to reach all its destinations within the same cycle. However, Section 6.8.4 shows that Prio=Bypass can lead to more false negatives, wasting bandwidth (since Prio=Bypass forces routers to wait for incoming bypassing flits at the cost of local flits), making a case for Prio=Local.

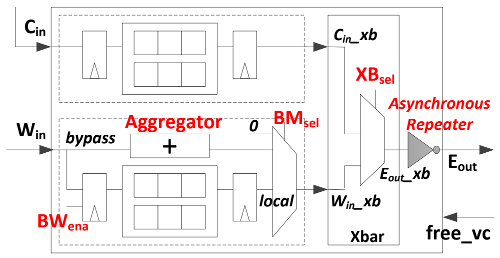
6.6 SMART for Many-to-1 traffic

In this section, we add aggregation support to SMART, to enable single-cycle chip-wide aggregation of Many-to-1 flows. We present two alternate designs tied to the two SSR priority schemes.

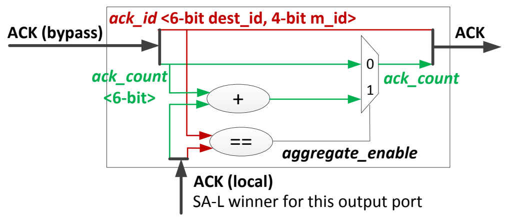
6.6.1 Aggregation with Prio=Local

We add the FANIN aggregation mechanism, presented earlier in Chapter 5, within each router. As before, the first ACK for a flow to get buffered at a router becomes the master ACK for that flow. It polls the latch at every input port, and aggregates any ACK corresponding to the same flow. Bypassing ACKs do not get aggregated. However, since Prio=Local prioritizes local flits over bypassing flits, a bypassing ACK will in fact stop if there is any local flit competing for the output link. Upon getting latched, this ACK will get aggregated if there is a master ACK for the same flow.

Suppose two ACKs A and B destined for Router R3 arrive at Routers R0 and R1 respectively in exactly the same cycle, and then send SSRs. Prio=Local will stop the ACK A at R1 and send ACK B from R1 to R3 - both in the same cycle. Subsequently ACK A will reach R3. This is a missed aggregation opportunity due to the priority scheme. To address this, we propose an alternate design next.

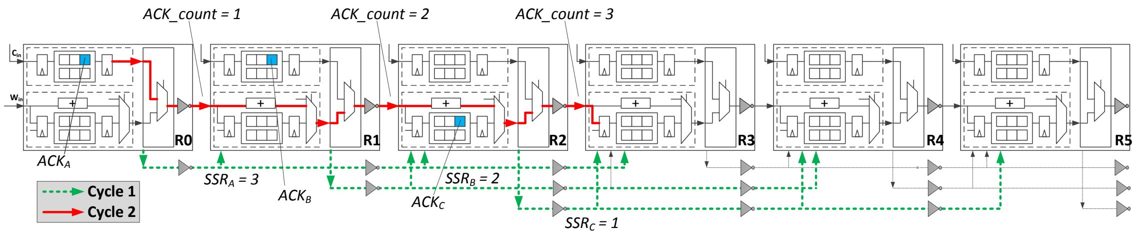


(a) **ACK Aggregator on bypass path.**



(b) **ACK Aggregator Circuit.**

Figure 6-12: ACK Aggregator in SMART Router.



BW_{ena}	0
BM_{sel}	0
XB_{sel}	C_{in}->E_{out}
Aggr	X

BW_{ena}	0
BM_{sel}	bypass
XB_{sel}	W_{in}->E_{out}
Aggr	1

BW_{ena}	0
BM_{sel}	bypass
XB_{sel}	W_{in}->E_{out}
Aggr	1

BW_{ena}	1
BM_{sel}	0
XB_{sel}	X
Aggr	X

BW_{ena}	0
BM_{sel}	0
XB_{sel}	X
Aggr	X

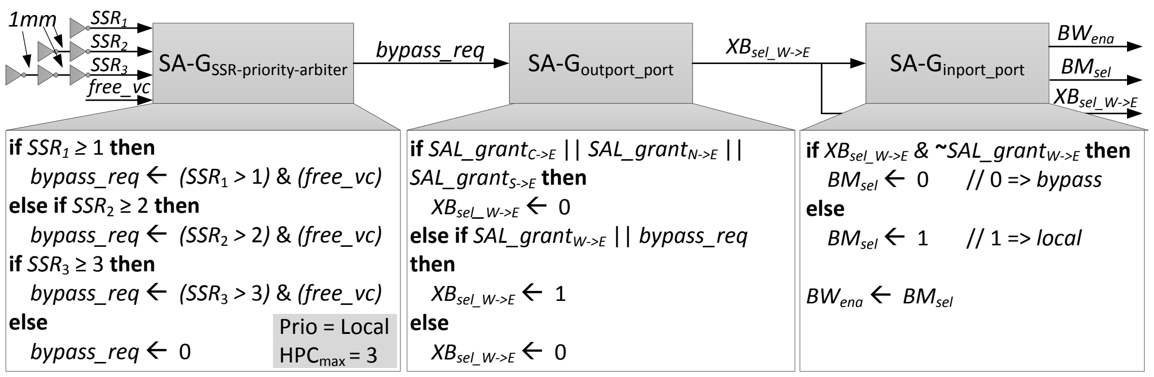
BW_{ena}	0
BM_{sel}	0
XB_{sel}	X
Aggr	X

Figure 6-13: M-to-1 Aggregation Example with Prio=Bypass.

6.6.2 Aggregation with Prio=Bypass

To avoid the missed aggregation opportunity just described, we propose a methodology to aggregate ACKs *while* an ACK is performing a multi-hop traversal. This design only works with Prio=Bypass. We add an *aggregator* module on the bypass path as shown in Figure 6-12(a). This aggregator module houses a comparator and an adder, as shown in Figure 6-12(b). During the ST+LT stage of a bypassing ACK, the comparator at every router compares the *ack_id* of the bypassing ACK with that of the SA-L winner for that output port (which is forced to wait because of the bypassing flit). As explained earlier in Section 5.5.2, the $ack\_id = [dest\_id, m\_id]$, where m_id is assigned by the controller that sent the preceding broadcast. In case of a match, the *ack_count* of the SA-L winner is aggregated into the *ack_count* of the bypassing ACK, and the SA-L winner is dropped. Figure 6-13 shows an example of a bypassing ACK from R0 to R3 aggregating the buffered ACKs at R1 and R2.

The main trade-off of this design is that aggregation is now on the critical path



```
if $SSR_1 \geq 1$ then
    bypass_req ← ($SSR_1 > 1$) & (free_vc)
else if $SSR_2 \geq 2$ then
    bypass_req ← ($SSR_2 > 2$) & (free_vc)
if $SSR_3 \geq 3$ then
    bypass_req ← ($SSR_3 > 3$) & (free_vc)
else
    bypass_req ← 0
```

```
if $SAL\_grant_{C\to E}$ || $SAL\_grant_{N\to E}$ ||
$SAL\_grant_{S\to E}$ then
    $XB_{sel\_W\to E}$ ← 0
else if $SAL\_grant_{W\to E}$ || bypass_req
then
    $XB_{sel\_W\to E}$ ← 1
else
    $XB_{sel\_W\to E}$ ← 0
```

```
if $XB_{sel\_W\to E}$ & ~$SAL\_grant_{W\to E}$ then
    $BM_{sel}$ ← 0      // 0 => bypass
else
    $BM_{sel}$ ← 1      // 1 => local

$BW_{ena}$ ← $BM_{sel}$
```

Figure 6-14: Implementation of SA-G at W_{in} and E_{out} (Figure 6-2) for SMART_1D.

of the multi-hop traversal, affecting HPC_{max}. For a 64-core system - 6-bit $dest\_id$ - and 4-bit m_id, we observe that the HPC_{max} of the design reduces from 11 to 6. Alternately, we can use a bit-vector for ACKs within each flit (potentially increasing the flit size), with each bit representing the core that sent the ACK. The 6-bit adder can now be replaced by a 64-bit OR gate which is much faster. However, we still need the 10-bit comparator that limits the critical path.

6.7 SMART Implementation

In this section, we describe the implementation details of the design, and discuss overheads. All numbers are for a k-ary 2-Mesh, i.e., the crossbar has 5-ports (with u-turns disallowed).

The SMART data-path, shown earlier in Figure 6-3, is modeled as a series of 128-bit 2:1 mux (for bypass) followed by a 4:1 mux (crossbar), followed by a 128-bit 1mm link.

The SMART control-path consists of HPC_{max}-hops repeated wire delay (SSR traversal), followed by logic gate delay (SA-G). In SMART_1D, each input port receives one SSR from every router up to HPC_{max}-hops away in that dimension. The logic for SA-G for Prio=Local in a SMART_1D design at the W_{in} and E_{out} ports of the router is shown in Figure 6-14[19]. The input and output signals correspond to the

[19]The implementation of Prio=Bypass is not discussed but is similar.

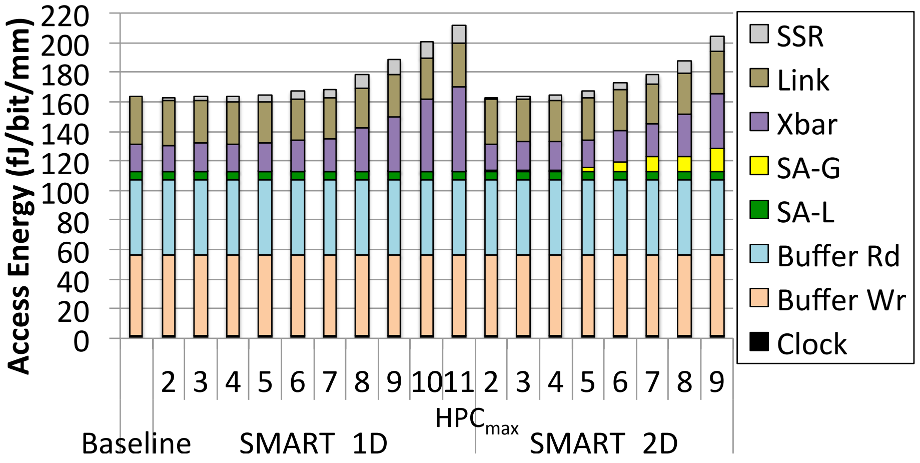


Figure 6-15: Energy/Access (i.e., Activity = 1) for each bit sent.

ones shown in the router in Figure 6-2[20].

In SMART_2D, all routers that are H-hops away, $H \in [1, HPC_{max}]$, together send a total of $(2 \times HPC_{max} - 1)$ $SSRs$ to every input port. SA-G$_{SSR\_priority\_arbiter}$ is similar to Figure 6-14 in this case and chooses a set of winners based on hops, while SA-G$_{output\_port}$ disambiguates between them based on direction, as discussed earlier in Section 6.4.2.

We choose a clock frequency of 1GHz based on SA-L critical path in the baseline 1-cycle router at 45nm [66]. We design each of the SMART components in RTL, run it through synthesis and layout for increasing HPC_{max} values, till timing fails at 1GHz[21]. This gives us energy and area numbers for every HPC_{max} design point. We incorporate these into energy and area numbers for the rest of the router components from DSENT [76]. Figure 6-15 plots the energy/bit/hop for accessing each component. For instance, if a flit wins SA-L and SA-G and traverses a SMART-hop of length 4 in an HPC_{max}=8 design, the energy consumed will be $E_{SA-L} + 8 \cdot E_{SSR} + 4 \cdot E_{SA-G} + E_{buf\_rd} + 4 \cdot E_{Xbar} + 4 \cdot E_{Link} + E_{buf\_wr}$.

The SMART data-path is able to achieve a HPC_{max} of 11. The extra energy

[20]To reduce the critical path, BW_{ena} is relaxed such that it is 0 only when there are bypassing flits (since the flit's valid-bit is also used to decide when to buffer), and BM_{sel} is relaxed to always pick local if there is no bypass. XB_{sel} is strict and does not connect an input to an output port unless there is a local or SSR request for it.

[21]The layout tool Cadence Encounter keeps the crosstalk optimization option, described in Appendix A.3.4, enabled in all these runs.

consumed by the repeaters for driving the bypass and crossbar muxes is part of the Xbar component in Figure 6-15, and increases comparatively insignificantly till about HPC_{max}=8, beyond which it shows a steep rise, consuming 3X of the baseline Xbar energy at HPC_{max}=11. The total data-path (Xbar+Link) energy for HPC_{max}=11 goes up by 35fJ/bit/hop, compared to the baseline. However, compared to the buffer energy (110fJ/bit/hop) that will be saved with the additional bypassing brought about by longer HPC_{max}, and coupled with additional network latency savings along with further reduction of data-path energy per bit as technology scales, we believe it will be worthwhile to go with higher HPC_{max} as we scale to hundreds or a thousand cores. The repeaters add negligible area overhead since they are embedded within the wire dominated crossbar.

SMART_1D's control-path is able to achieve a HPC_{max} of 13 (890ps SSR, 90ps SA-G). But the overall HPC_{max} gets limited to 11 by the data-path. SA-G adds less than 1% of energy or area overhead.

SMART_2D's control-path is able to achieve a HPC_{max} of 9 (620ps SSR, 360ps SA-G), at which point the energy and area overheads go up to 8% and 5% respectively, due to the quadratic scaling of input $SSRs$ with HPC_{max}. However, not all the input SSRs are likely to be active every cycle.

The total number of SSR-bits entering an input port are of $O(HPC_{max} \cdot log_2(HPC_{max}))$ and $O({HPC_{max}}^2 \cdot log_2(HPC_{max}))$ in SMART_1D and SMART_2D respectively. But these do not affect tile area. However, the $SSRs$ add energy overheads due to HPC_{max}-mm signaling whenever a SSR is sent.

Based on the energy results, we choose HPC_{max}=8 for both SMART_1D and SMART_2D for our evaluations. For SMART_1D, HPC_{max}=8 allows bypass of all routers along the dimension and the ejection, in our target 8-ary 2-Mesh.

6.8 Evaluation

In this section, we evaluate SMART with both synthetic traffic, as well as full-system traffic. Our target system is shown in Table 6.2. The baseline design in all our runs,

Table 6.2: Target System and Configuration

Process		**On-chip Network**	
Technology	45nm	Topology	8-ary 2-Mesh
V_{dd}	1.0 V	Router Ports	5
Frequency	1.0 GHz	Routing	XY
Link Length	1mm	Flit Width	128-bit

Synthetic Traffic	
Virtual Channels	12 [1-flit/VC]

Full-system Traffic	
Processors	64 in-order SPARC
L1 Caches	Private 32kB I&D
L2 Caches	Private/Shared 1MB per core
Cache Coherence	MOESI distributed directory
Virtual Networks	3 (req, fwd, resp)
Virtual Channels	4 (req) [1-flit/VC], 4 (fwd) [1-flit/VC], 4 (resp) [5-flit/VC]

BASELINE (t_r=1), is a state-of-the-art NoC with 1-cycle routers. For experiments with multicast and reduction traffic, we use FANOUT and FANOUT+FANIN as the baseline designs. All SMART designs are named as **SMART-**HPC_{max}**_1D/2D**. Prio=Local is assumed[22], unless explicitly mentioned. We first evaluate SMART with 1-to-1 unicast traffic (synthetic and full-system) and then move on to evaluations with 1-to-Many and Many-to-1 traffic.

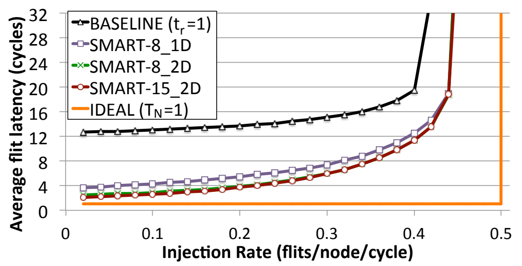
6.8.1 Synthetic 1-to-1 Traffic

SMART across different traffic patterns.

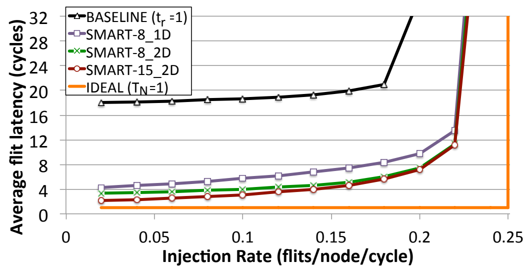
We start by running SMART with different synthetic traffic patterns, as shown in Figure 6-16. We compare 3 SMART designs: SMART-8_1D and SMART-8_2D (which are both achievable designs as discussed in Section 6.7), and SMART-15_2D which reflects the best that SMART can do in an 8×8 Mesh (with maximum possible hops = 15). We inject 1-flit packets to first understand the benefits of SMART without secondary effects due to flit serialization, and VC allocation across multiple routers etc. For the same reason, we also give enough VCs (12, derived empirically) to allow both the baseline and SMART to be limited by links, rather than VCs for throughput.

The striking feature about SMART from Figure 6-16 is that it pushes low-load

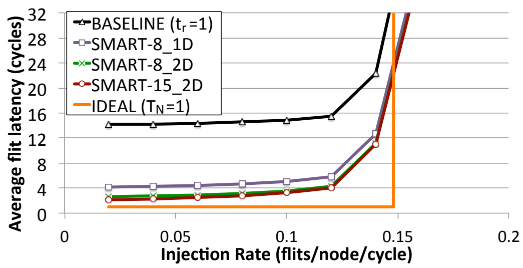
[22]The reason will become clear in Section 6.8.1 where we compare Prio=Local and Prio=Bypass.



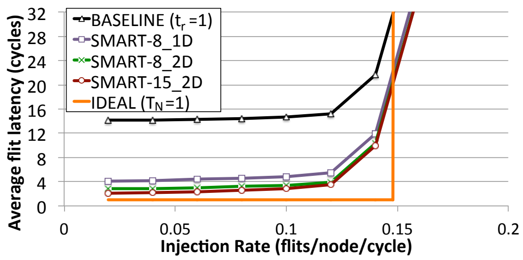
(a) **Uniform Random (Avg Hops = 5.33).**



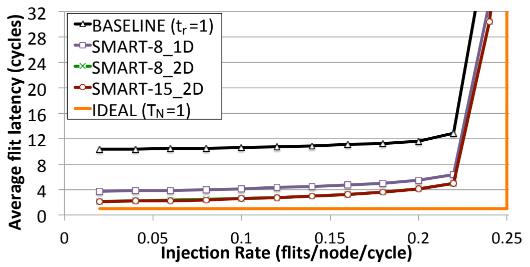
(b) **Bit Complement (Avg Hops = 8).**



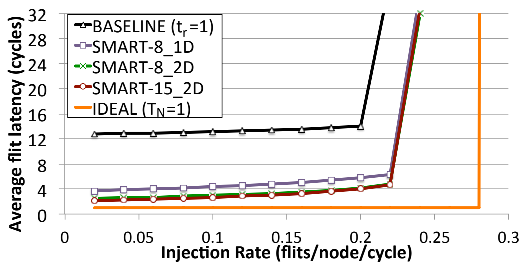
(c) **Bit Reverse (Avg Hops = 6).**



(d) **Transpose (Avg Hops = 6).**



(e) **Shuffle (Avg Hops = 4).**



(f) **Hot Spot 5%* (Avg Hops = 5.25).**

*Uniform Random traffic, where router at center of edge receives 5% more traffic than others.

Figure 6-16: SMART with synthetic unicast traffic.

latency to 4 and 2 cycles, for SMART_1D and SMART_2D respectively, across all traffic patterns, unlike the baseline where low-load latency is a function of the average hops, thus truly breaking the locality barrier. SMART-8_2D achieves most of the benefit of SMART-15_2D for all patterns, except Bit Complement, since average hop counts are ≤ 8 for an 8×8 Mesh. SMART also increases network throughput by 7-13% in Uniform Random, Bit Complement and Hot Spot traffic scenarios.

Impact of HPC_{max}.

Next we study the impact of HPC_{max} on performance. We plot the average flit latency for Bit Complement traffic (which has high across-chip communication) for

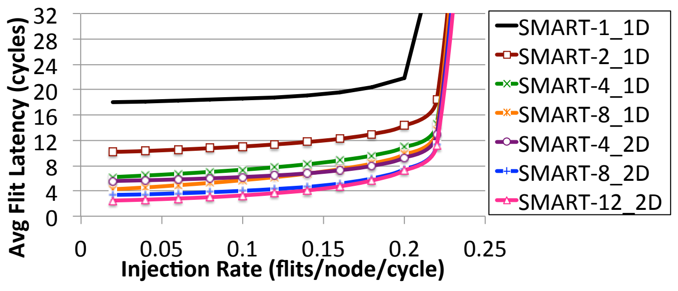
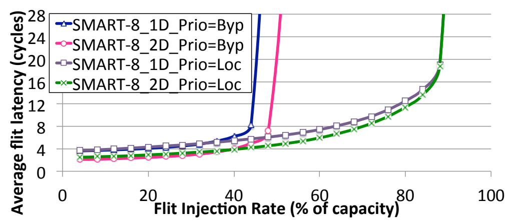


Figure 6-17: Impact of HPC_{max} (Bit Complement).

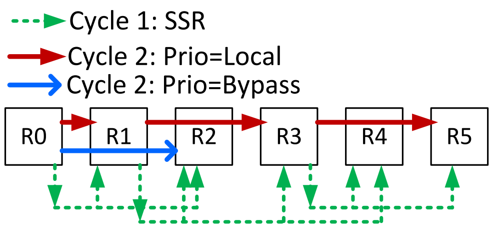
HPC_{max} from 1 to 12, across 1D and 2D in Figure 6-17. SMART-1_1D is identical to Baseline (t_r=1) as it does not need SA-G. HPC_{max} of 2 itself gives a 1.8X low-load latency reduction, while 4 gives a 3X reduction. These numbers indicate that even with a faster clock, say 2.25GHz, which will drop HPC_{max} to 4, a SMART-like design is a better choice than a 1-cycle router. It should also be noted that as we scale to smaller feature sizes, cores shrink while die sizes remain unchanged, so the same SMART interconnect length will translate to larger HPC_{max}. Adding SMART_2D, and increasing HPC_{max} to 12 pushes low-load latency close to a 2-cycles: an 8.4X reduction over the baseline. This result highlights that a heavily-pipelined higher frequency baseline can only match a 1GHz SMART NoC if it runs at 8.4GHz.

Impact of SA-G priority.

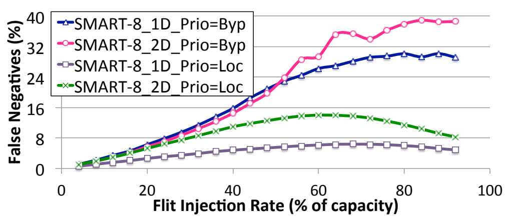
We study the effects of priority in Figure 6-18(a) for the best possible 1D and 2D SMART designs. While both priority schemes perform identically at very low-loads, Prio=Bypass has a sudden throughput degradation at an injection rate of about 44-48% of network capacity. Intuitively, we would expect Prio=Bypass to be better than Prio=Local as it allows for longer bypass paths, and avoids unnecessary stopping and buffering of flits already in flight. Moreover, it is often the priority scheme used in non-speculative 1-cycle router designs [51, 56] when choosing between a lookahead and a local buffered flit. However, for SMART, where multiple allocations are occurring in the same cycle, it suffers from a unique problem, highlighted in Figure 6-18(b). In this example, Router's R0, R1 and R3 send $SSRs$ up to R2, R4 and R5 respectively, in Cycle-1. In a Prio=Local scheme, R0's SSR would lose at R1, and R1's SSR



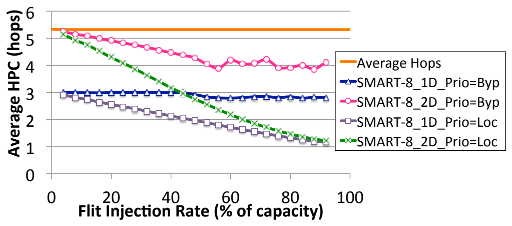
(a) **Average Network Latency.**



(b) **Throughput Loss in Prio=Bypass.**



(c) **Percentage of False Negatives.**



(d) **Average achievable HPC.**

Figure 6-18: Prio=Local vs. Prio=Bypass for Uniform Random Traffic.

would lose at R3, leading to the traversals shown in Cycle-2. For Prio=Bypass, R0's SSR will win at R1, and the corresponding flit will be able to go all the way to its stop router R2. However, even though R1's SSR lost SA-G at its start router, it wins over R3's SSR at R3, preventing R3 from sending its own flit. This cascading effect can continue, leading to forced starvation of flits (i.e., flits are not allowed to use the output link even though it is idle) and poor link utilization, causing heavy throughput loss. This effect is reflected in the percentage of false negatives (cases where a router was expecting a flit but no flit came) going up to 25-40% in Prio=Bypass, killing its throughput, as opposed to less than 10% in Prio=Local[23] as shown in Figure 6-18(c). On the plus side, Prio=Bypass always creates SMART-hops with high HPCs, since a flit that starts only stops at its requested stop router, or at the turn router in this priority scheme. This can be seen in Figure 6-18(d) where SMART-8_1D_Prio=Bypass achieves an average HPC of 3, while SMART-15_2D_Prio=Bypass maintains an HPC of 4-5. Prio=Local, on the other hand, drops the achievable HPC to 1 at high loads.

[23]False negatives in Prio=Local do not cause any throughput loss. They occur if downstream routers receive an SSR and set up a bypass path but no flit arrives as it was prematurely stopped at an upstream router to make way for a local flit that is turning or getting ejected into the NIC. Prio=Local always ensures that a local flit gets to use the output link without any starvation. On the other hand, false negatives in Prio=Bypass cause harm because routers block their local flits from using the output link if they are expecting a bypass flit.

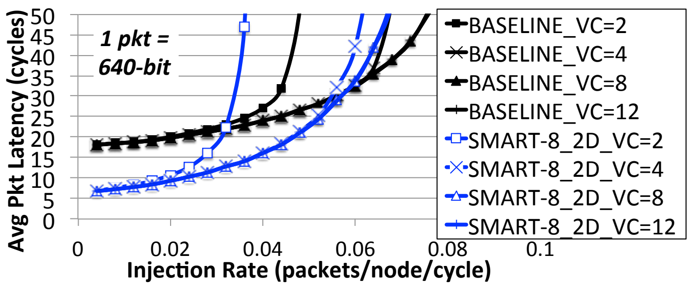


Figure 6-19: Impact of 5-flit packets (Uniform Random).

Impact of multi-flit packets.

SMART locks an input and output port till all flits of a packet leave, to implement virtual cut-through (Section 6.3.4). Thus, it artificially suffers from the head-of-line blocking problem. In other words, congestion at the output port can block other packets at the input port as well, throttling throughput. In the baseline VC router, on the other hand, flits from other packets can use the input bandwidth for other output ports. A second problem with multi-flit packets comes because of ordering. Flits might need to be pre-maturely buffered to avoid the Body/Tail overtaking the Head/Body (Section 6.3.3). This lowers output link utilization, lowering throughput. A third problem is that multi-flit packets reserve VCs at all routers along a multi-hop path to handle premature stopping (Section 6.3.4). These effects are visible in Figure 6-19 which evaluates SMART with Uniform Random traffic where all packets have 5-flits - a worse case adversarial traffic scenario. SMART has 11% lower throughput than the baseline, even with 12 VCs.

Comparison with High-Radix Topology.

We compare SMART with a Flattened Butterfly [48] topology. Each FBfly router has dedicated single-cycle links to every other node in that dimension (i.e., radix-15). We assume that the router delay is 1-cycle. This is a very aggressive assumption, especially because the SA stage needs to perform 22:1 arbitrations. All high-radix routers assume > 4-cycle pipelines [48, 49, 74]. We use 8 VCs per port with virtual cut-through in both SMART and FBfly (thus giving more buffer resources to FBfly). In Figure 6-20, we plot three configurations where the total number of wires, i.e.,

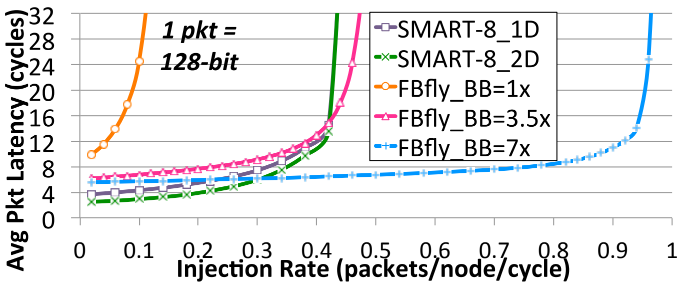


Figure 6-20: SMART vs. Flattened Butterfly (Uniform Random).

Bisection Bandwidth (BB), of the FBfly is 1x, 3.5x and 7x that of SMART (leading to 7-flits, 2-flits and 1-flit per packet respectively for 128-bit packets). At BB=1x, FBfly loses both in latency and throughput due to heavy serialization delay. At BB=3.5x, FBfly can match SMART in throughput. Despite an aggressive 1-cycle router, at BB=7x the best case latency for FBfly is 6 cycles (2 at injection, 2 at turning, and 2 at ejection router) as compared to 4 and 2 for SMART-1D and SMART-2D respectively. The radix-15 FBfly_BB=3.5x router, modeled in DSENT [76], incurs an area, dynamic power (at saturation) and leakage power overhead of 3.9x, 1.5x and 10x respectively over SMART. If we are willing to use N times more wires, a better solution would be to just have N meshes, each with SMART, so that fast latency is achieved in addition to reconfigurable bandwidth.

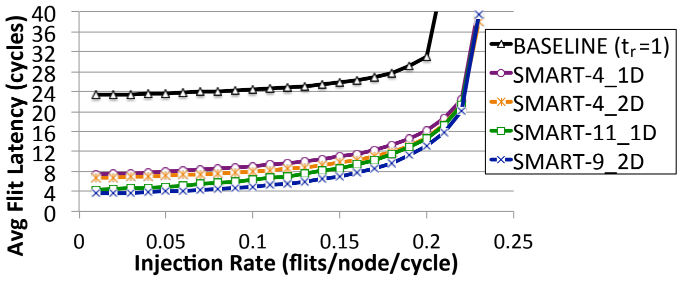
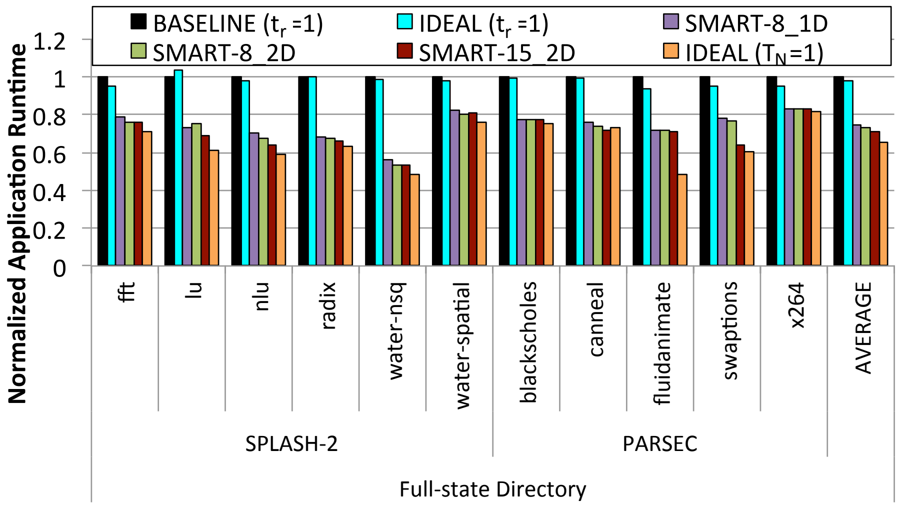


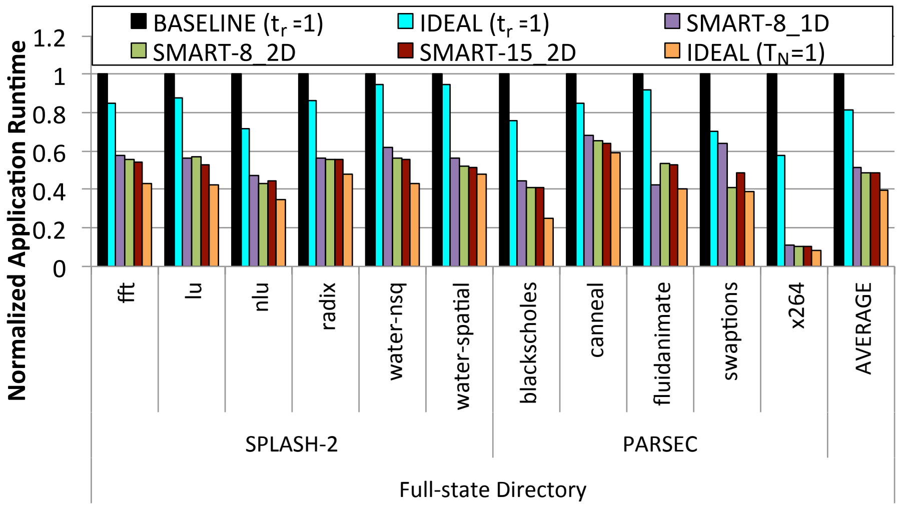
Figure 6-21: SMART on a 256-core mesh (Uniform Random).

SMART on a 16×16 mesh.

Figure 6-21 plots the performance of SMART on a 256-core mesh with Uniform Random traffic. SMART scales well, with HPC_{max}=4 lowering network latency from 23 to 6-7 cycles at low loads. SMART-11_1D and SMART-9_2D lower it even further



(a) **Private L2.**



(b) **Shared L2.**

Figure 6-22: Full-system application runtime for full-state directory protocol, normalized to BASELINE (t_r=1).

to 3-4 cycles. SMART also gives a 12% throughput improvement.

6.8.2 Full-system Directory Protocol Traffic

We evaluate the parallel sections of SPLASH-2 [82] and PARSEC [14] running a full-state directory protocol for both *Private L2* and *Shared L2* implementations. Table 6.2 describes the configuration of our system. In Private L2 designs, a copy of the data is retained in the local L2 within the tile for fast access upon future L1 misses. All

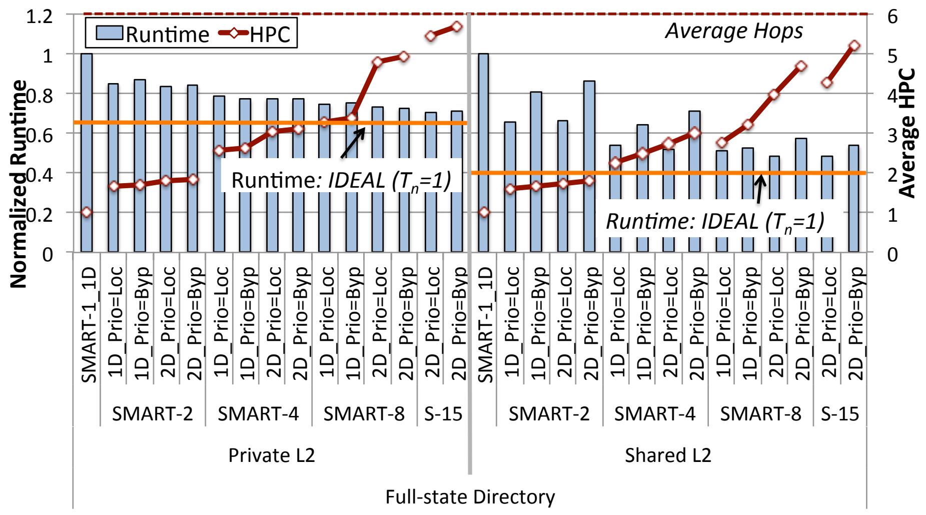


Figure 6-23: Impact of HPC_{max} and Priority, averaged across all benchmarks.

L1 misses first lookup the local L2, before forwarding the request to the directory. However, replication of data across tiles lowers the effective on-chip L2 cache capacity. In Shared L2 designs, there is only one L2 copy of the data on-chip, increasing L2 cache capacity. However, L1 misses always involve a network traversal to access data at the remote L2, making on-chip latency critical to performance.

Performance Impact.

Figure 6-22 shows that SMART-8_1D and SMART-8_2D lower application runtime by 26% and 27% respectively on average, for a Private L2, which is only 8% away from the IDEAL (T_N=1). The runtime reduction goes up to 49% and 52% respectively with a Shared L2 design, which is 9% off from the IDEAL (T_N=1). SMART-15_2D does not give any significant runtime benefit over SMART-8_2D.

Impact of HPC_{max} and SA-G priority.

Figure 6-23 sweeps through HPC_{max} and SA-G priority, and plots the normalized runtime and achieved HPC, on average across all the benchmarks. Since these full-system traffic fall in the lower end of the injection rates in the synthetic traffic graphs, Prio=Bypass performs almost as well as Prio=Local, except at low HPC_{max} in a Shared L2. HPC_{max} of 4 suffices to achieve most of the runtime savings.

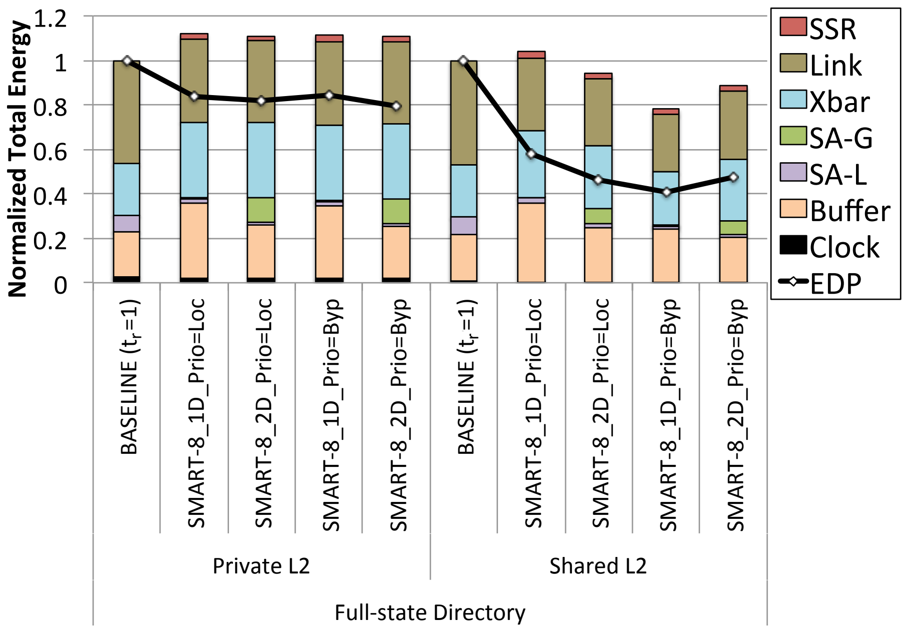
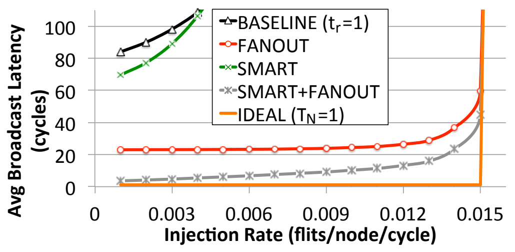


Figure 6-24: Total Network Dynamic Energy.

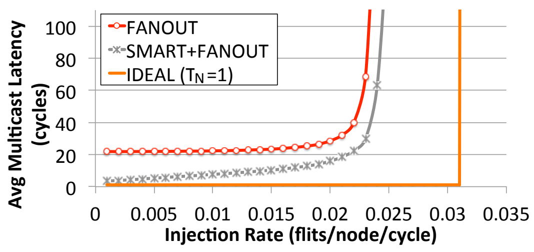
Total Network Energy.

Figure 6-24 explores the energy trade-off of SMART by plotting the total dynamic energy of the network consumed for running the benchmarks to completion, on average across all the benchmarks. For Private L2, the dynamic energy for SMART goes up by 10-12% across designs primarily due to the data-path, though the overall EDP goes down by 20%. For Shared L2, the dynamic energy goes down by 6-21% across the designs, because of a lower runtime. The EDP goes down by up to 59%. SMART with Prio=Local consumes 18-46% higher energy in the buffers than both SMART with Prio=Bypass and the baseline (which also prioritizes incoming flits over already buffered local flits in SA to reduce buffering), since Prio=Bypass, by definition, reduces the number of times flits need to stop and get buffered. SA-G energy contributes less than 1% of network energy for SMART_1D, and goes up to about 10% for SMART_2D. All these ups and downs are however negligible when we also consider leakage. We observed leakage to contribute more than 90% of the total energy, since the network activity is very low in full-system scenarios[24]. However, even

[24]Leakage could be reduced by aggressive power gating solutions, which itself is a research challenge.



(a) **Uniform Random Broadcast (1-to-Many, Many: 64).**



(b) **Uniform Random Multicast (1-to-Many, Many: 2-64).**

Figure 6-25: SMART+FANOUT with synthetic uniform random 1-to-Many traffic.

with high leakage, the total network power was observed to be about 3W for both baseline and SMART, while chip power budgets are usually about 100W. Thus the energy overheads of SMART are negligible.

6.8.3 Synthetic 1-to-Many Traffic

We evalauate the SMART+FANOUT design with synthetic broadcast and multicast traffic. All SMART networks in this subsection implement SMART-8_2D to represent the SMART design with the lowest unicast latency.

Broadcast-only Traffic.

Figure 6-25(a) plots the performance of the baseline (t_r=1), FANOUT, and SMART networks with broadcast-only traffic. All sources inject broadcasts at the specified injection rate in a uniform random manner. The metric of comparison is the average broadcast latency, i.e., the average of the number of cycles it takes for all NICs on the chip to receive the broadcast. The baseline (t_r=1) and SMART networks are fork@nic networks, i.e., the broadcast is implemented as 64 unicasts being sent out from the source. This adds heavy serialization increasing the low-load latency and saturating these networks at only 25% of the IDEAL (T_N=1) capacity. With FANOUT, which was presented in Chapter 4, the network adds support for forking flits at routers, with t_r=1, allowing the broadcast to be received by all nodes in 23 cycles at low loads. This network pushes throughput to almost the same as the IDEAL (T_N=1). Enhancing SMART with FANOUT, i.e., support for forking flits at routers, pushes the low-load broadcast latency to 3.6 cycles, a reduction of 84% compared to FANOUT. The low-load latency of SMART+FANOUT with broadcast traffic is only 1.6 cycles higher than the 2-cycle low-load latency for SMART with unicast traffic, which means that sending a broadcast is no longer an expensive high-latency operation.

Multicast Traffic.

Figure 6-25(b) plots the performance of FANOUT, SMART+FANOUT and the IDEAL (T_N=1) networks with multicast traffic, where the source randomly picks a destination set comprising anywhere from 2 to 64 (all) nodes. FANOUT saturates at a throughput of 76% of the ideal and SMART+FANOUT pushes it to 80%. At low loads, SMART+FANOUT distributes the multicast to all its destinations in 3.4 cycles on average, as compared to 21 cycles with just FANOUT.

6.8.4 Full-system Broadcast Protocol Traffic

We run SPLASH-2 and PARSEC on systems implementing Token Coherence (no directory) and HyperTransport (no-state directory). Both designs use a Private L2 implementation. In Chapter 2 we saw that 52.4% of all injected messages in Token Coherence are broadcasts, while 14.3% and 14.1% of all injected messages are broadcasts and acknowledgements respectively in HyperTransport. FANOUT described in Chapter 4 was shown to address the latency and throughput requirements of 1-to-Many (multicast) flows, while FANIN in Chapter 5 was shown to address the latency and throughput requirements of Many-to-1 (acknowledgement) flows. Here we evaluate the impact of FANOUT and FANIN enhanced with SMART. Unless specified otherwise, SMART-8_2D is implemented to represent the best performing and implementable (at 45nm) SMART NoC.

Performance Impact.

Figure 6-26 plots the full-system application runtime for Token Coherence with a variety of network implementations. Compared to FANOUT - routers with t_r=1

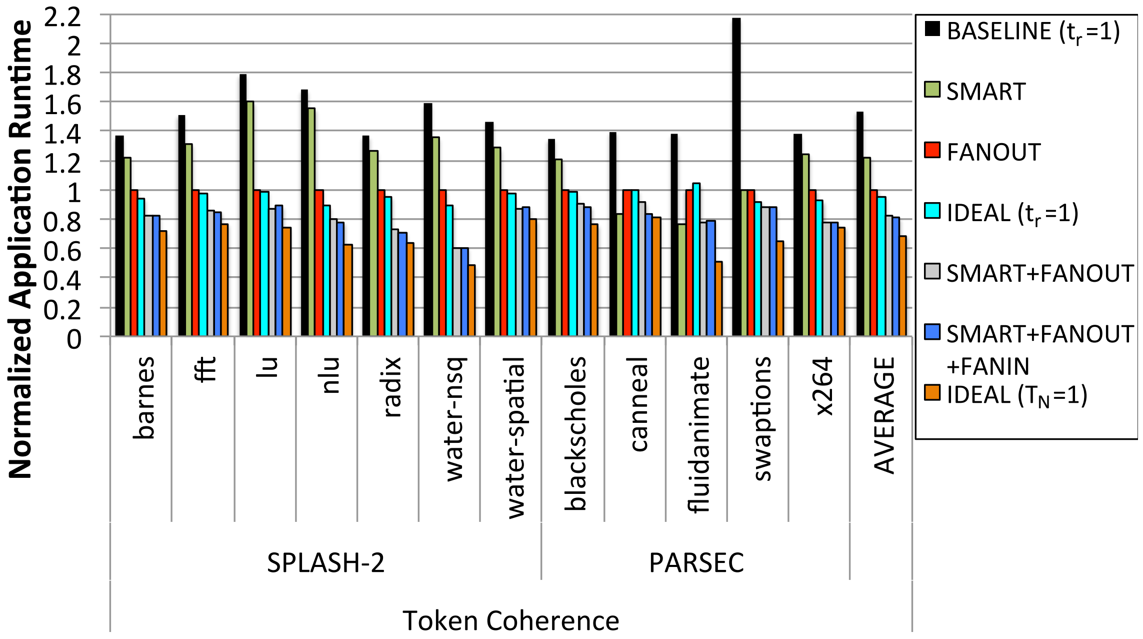


Figure 6-26: Full-system application runtime for Token Coherence, normalized to FANOUT.

and multicast support -, BASELINE (t_r=1) and SMART - both without any multicast support - are 54% and 22% slower on average, respectively. SMART+FANOUT improves performance by 18% compared to FANOUT. Adding FANIN improves performance by a further 1%. The design is 12% away from the IDEAL (T_N=1) NoC.

Figure 6-27 plots the full-system application runtime for HyperTransport. Here, as observed earlier in Chapter 4, FANOUT alone does not provide much benefit since the M-to-1 ACKs saturate the network and determine the runtime. Compared to FANOUT+FANIN - routers with t_r=1 and multicast + aggregation support -, SMART - without any multicast or aggregation support - is 19% slower on average. SMART+FANOUT - FANOUT with single-cycle multihop paths for both unicasts and broadcasts - lowers runtime of FANOUT by 14%, but is still 5% away from FANOUT+FANIN. This is because ACKs cannot leverage the advantage of SMART because the traffic is hot-spot to one destination in a M-to-1 flow. SMART is targeted to lower latency, but cannot help the bandwidth limited ACKs which contend for the same set of links. SMART+FANOUT+FANIN which adds aggregation support[25]

[25]Here we plot SMART+FANOUT+FANIN with Prio=Local. The impact of Prio=Local vs. Prio=Bypass for SMART+FANOUT+FANIN will be discussed later in this section.

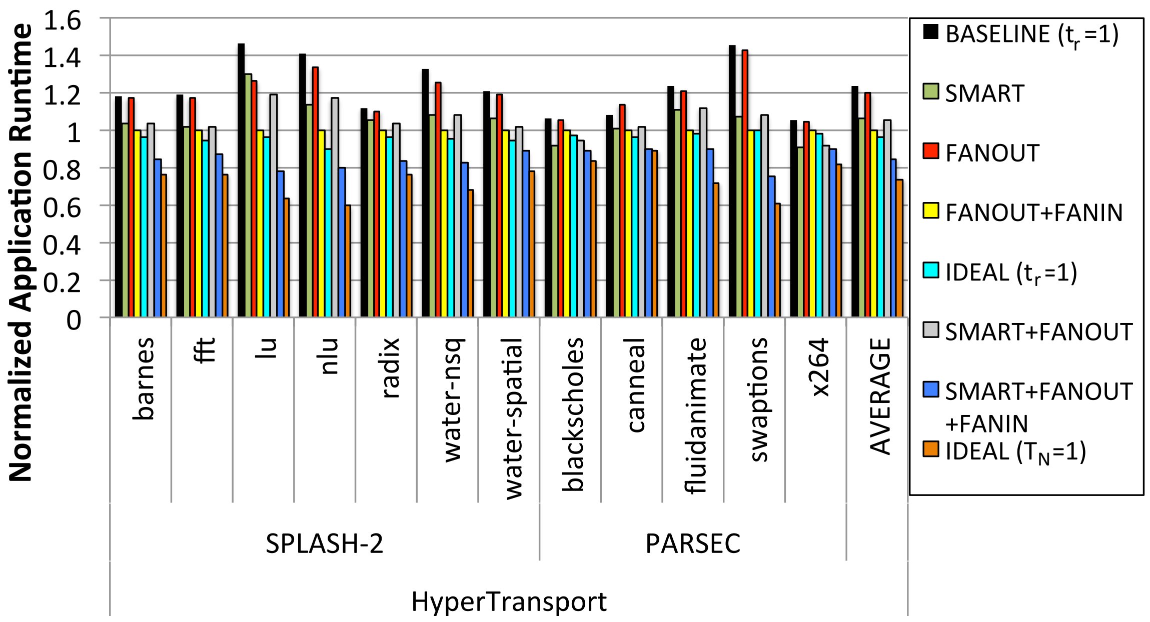


Figure 6-27: Full-system application runtime for HyperTransport, normalized to FANOUT+FANIN.

to SMART+FANOUT helps lower runtime by 15% compared to FANOUT+FANIN. The design is 11% away from the IDEAL (T_N=1) NoC.

It needs to be noted that the IDEAL (T_N=1) assumes unlimited bandwidth, while the realistic SMART+FANOUT+FANIN needs to serialize flows over the limited shared links of a mesh. The SMART NoC is able to reduce runtime by 15-27% compared to the respective baseline single-cycle per-hop networks with multicast and aggregation support, across the three coherence protocols, and is 8-12% away from an ideal network which always offers a single-cycle network delay.

Average Broadcast Latency.

Figure 6-28 shows the average broadcast latency across the benchmarks. For Token Coherence, average broadcast latency with SMART+FANOUT is 9.3 cycles, and goes down to 8.7 cycles with the addition of FANIN. Though this is a 64% reduction compared to FANOUT+FANIN, contention with other broadcast and unicast flows in the network, and no special priority for broadcast flits in the arbiters, means that broadcasts do not get delivered to all nodes in 3.6 cycles like we saw in Figure 6-25(a).

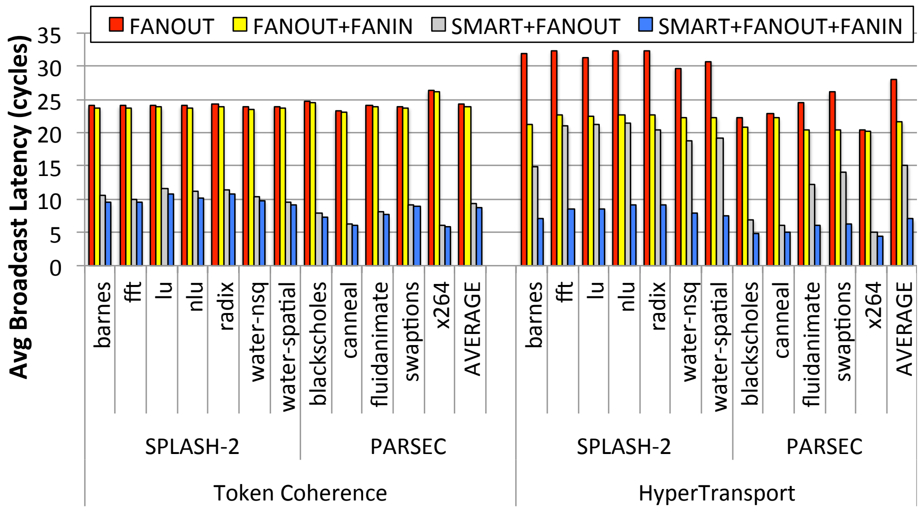


Figure 6-28: Average Broadcast Latency.

For HyperTransport, SMART+FANOUT lowers the average broadcast latency from 22 to 15, while SMART+FANOUT+FANIN lowers it to 7 cycles.

Impact of HPC_{max} and SA-G Priority.

Figure 6-29 shows the impact of HPC_{max} and SA-G Priority on the runtime, ratio of ACKs aggregated, and percentage of false negatives; averaged across the SPLASH-2 and PARSEC benchmarks with HyperTransport. SMART-15 is also plotted to present the best that SMART can do in a 8×8 mesh, even though HPC_{max}=15 is not realizable at 45nm.

The first observation we make is that beyond HPC_{max}=4 we do not see a significant change in performance. This means that the reducion of HPC_{max} to 6 for SMART+FANIN with Prio=Bypass does not have any adverse effect on performance.

SMART+FANIN aggregates 81.5% of ACKs with Prio=Local, and a further 8.5% with Prio=Bypass. With Prio=Bypass, 32% of the ACKs get aggregated in the buffers, and 58% of the ACKs are aggregated by the bypassing flits. However, this does not translate to any significant runtime improvement for Prio=Bypass compared to Prio=Local. Both priority schemes have similar runtime.

At HPC_{max}=4, the percentage of false negatives in Prio=Bypass jumps to 7.5%, and goes up to and stays at 8.8% beyond HPC_{max}=8. For Prio=Local, this number

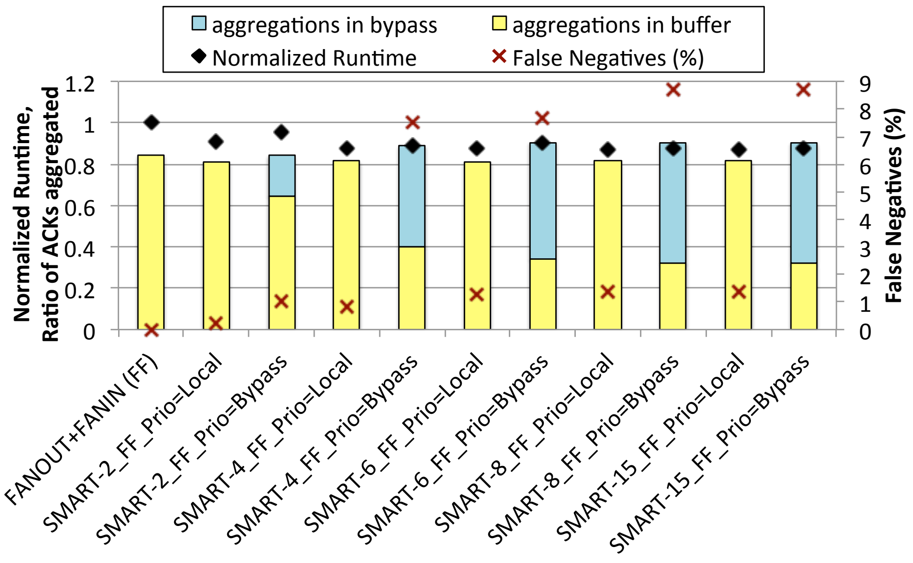


Figure 6-29: Impact of HPC_{max} and SA-G Priority on runtime, ratio of ACKs aggregated, and false negatives for HyperTransport. All results are averaged across the SPLASH-2 and PARSEC benchmarks, and normalized to FANOUT+FANIN.

stays below 1.4%. The false negatives, which translate to lost throughput, explain why Prio=Bypass does not perform better than Prio=Local despite being better suited intuitively for both broadcasts and aggregations.

6.9 Related Work

High-radix routers. High-radix router designs such as CMesh [13], Fat Tree [22], Flattened Butterfly [48], BlackWidow [74], MECS [32], Clos [44] are topology solutions to reduce average hop counts, and advocate adding physical express links between distant routers. These express point-to-point links can be further engineered for lower delay with advanced signaling techniques like equalization [45] and capacitively-driven links [51]. Each router has > 5 ports, and link bandwidth (b) is often reduced proportionally to have similar buffer and crossbar area/power as a mesh (radix-5) router. More resources however imply a complicated routing and Switch+VC allocation mechanism, with a hierarchical SA and crossbar [49], increasing router delay t_r to 4-5 at the routers where flits do need to stop. These designs also complicate layout since multiple point-to-point global wires need to span across the chip. Moreover, a topology solution only works for certain traffic, and incurs higher latencies for adversarial traffic (such as near neighbor) because of higher serialization delay.

In contrast, SMART provides the illusion of dedicated physical express channels, embedded within a regular mesh network, without having to lower the link bandwidth, or increase the number of router ports.

Asynchronous NoCs. Asynchronous NoCs [17, 12] have been proposed for the SoC domain for deterministic traffic. Such a network is programmed statically to preset contention-free routes for QoS, with messages then transmitted across a fully asynchronous NoC (routers and links). Instead, SMART couples *clocked* routers with asynchronous links, so the routers can perform fast cycle-by-cycle reconfiguration of the links, and thus handle general-purpose CMPs with non-deterministic traffic and variable contention scenarios. Asynchronous Bypass Channels [39] target chips with multiple clock domains across a die, where each hop can incur significant synchronization delay. They aim to remove this synchronization delay. This leads them to propose sending a clock signal with the data so that the data can be latched correctly at the destination router. Besides this difference in link architecture, the different goals also lead to distinct NoC architectures. Due to the multiple clock domains, ABC needs to buffer/latch flits at every hop speculatively, discarding them thereafter if flits successfully bypassed. Also, the switching between *bypass* and *buffer* modes cannot be done cycle-by-cycle, which increases latency. In contrast, SMART targets a single clock domain across the entire die, so SSRs can be sent in advance to avoid latching flits at all along a multi-hop path and allow routers to switch between *bypass* and *buffer* modes each cycle.

6.10 Chapter Summary

In this chapter, we presented SMART, a NoC architecture that allows flits to dynamically setup and traverse multi-hop paths in a single-cycle, without adding any physical channels on the data-path. We replace clocked drivers at every router with repeaters, thus creating a repeated link, which has been shown to go up to 13-19mm at 1GHz in Appendix A. We also presented a flow control technique to opportunistically reserve multiple link segments within a cycle, and then allow a flit to virtually bypass multiple routers within a cycle and only get latched at the destination, or upon a conflict with another flow. SMART breaks the fundamental dependence of latency on H, the number of hops, thereby breaking the on-chip latency barrier. We also enhanced SMART to support forking and aggregation, thus enabling single-cycle chip-wide 1-to-Many and Many-to-1 traversals.

As technology scales, repeated global wire delay is expected to remain fairly constant. Since clock frequencies have plateaued due to the power wall, this means that repeated wire delay in clock cycles will remain the same. Meanwhile, at smaller feature sizes, cores (i.e., tiles) shrink but our chip dimensions remain similar due to yield limitations. All these trends point to an increasing HPC_{max} at future technologies, making SMART even more attractive. This work opens up a plethora of research opportunities in circuits, NoC architectures, many-core architectures and software to optimize and leverage SMART NoCs. We see SMART paving the way for locality-oblivious CMPs, easing the burden on coherence protocol and/or software from actively optimizing for locality.

7 Conclusions

What we call the beginning is often the end.
And to make an end is to make a beginning.
The end is where we start from.
- T. S. Elliot, *Four Quartets*

As we continue to scale multicore designs, the on-chip communication latency becomes critical for performance. Traditionally, the latency of multi-hop network traversals has been proportional to the number of hops times the router delay. Research in low-latency on-chip networks over the past decade has resulted in the router delay dropping to one cycle, enabling single-cycle per-hop networks. However, the delay in these networks grows proportional to the number of hops, which in turn increase as core counts scale.

The contribution of this thesis is a network-on-chip architecture that presents single-cycle traversal paths to messages over an underlying shared network fabric, for both one-to-one and collective (one-to-many and many-to-one) communication flows.

In this chapter, we walk through a summary of the main contributions of this thesis in Section 7.1. We conclude by examining future research directions in Section 7.2.

7.1 Dissertation Summary

This thesis proposed techniques to progressively design a network that facilitates single-cycle traversals across the chip, for a variety of traffic flows (1-to-1 or unicast,

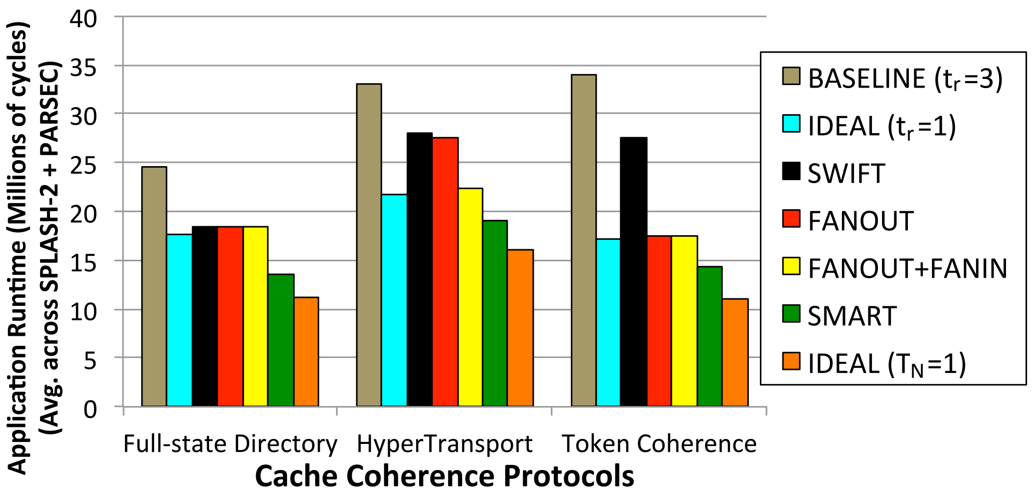


Figure 7-1: Thesis Contributions.

1-to-Many or multicast, and Many-to-1 or reduction), starting with a single-cycle per-hop network for unicast flows as the baseline.

A visual summary of the contributions is shown in Figure 7-1 which plots the average runtime across a suite of SPLASH-2 and PARSEC benchmarks running on a 8×8 Mesh CMP with the different NoC architectures proposed in this dissertation.

We started with the microarchitecture of a single-cycle router, and prototyped the same within a 2x2 NoC in 90nm. This design was called the SWIFT NoC. Each flit is preceded by a *lookahead* traveling one-cycle in advance and setting up the crossbar switch, allowing the incoming flit to bypass the conventional multi-stage router pipeline (buffering, followed by arbitrations for the switch and the next router's virtual channels) and proceed directly to the outgoing link via the crossbar. We also added *tokens* to this design which are hints about buffer availability in each router's neighborhood, allowing it to traverse adaptive congestion-free paths. The SWIFT NoC provides a 39% reduction in low-load latency for uniform random traffic and a 26% reduction in runtime across SPLASH-2 and PARSEC benchmarks running on a shared memory CMP with a full-state directory protocol.

For protocols with 1-to-Many (multicast) traffic flows, a single-cycle per-hop NoC like SWIFT does not provide much benefit because of heavy serialization at the source NIC (BASELINE_fork@nic) due to M flits being sent out for every 1-to-M

request. Prior attempts to add support within routers to fork multicast flits (BASELINE_fork@rtr) reduces the serialization problem at the source NIC (since only one flit enters the network), but at the cost of higher delays at each router. We proposed FANOUT: a series of optimizations comprising a load-balanced routing algorithm for multicasts, a crossbar circuit for efficient forking of multicast flits within a cycle at low energy overhead, and a single-cycle router microarchitecture. FANOUT speeds up broadcast delivery by 61%, and improves throughput by 63% compared to the fork@rtr design. It reduces full-system runtime by 10% on average across SPLASH-2 and PARSEC benchmarks running on a shared memory CMP with the broadcast-intensive Token Coherence protocol.

For protocols with Many-to-1 (reduction) traffic flows (such as acknowledgements), heavy serialization at the destination NIC (due to M flits being delivered for every M-to-1 response) and congestion at links leading up to the destination hot spot router lead to early network saturation and system slowdown. We proposed FANIN: a series of optimizations comprising a load-balanced routing algorithm for reduction traffic, a flow control technique to identify and aggregate ACKs with routers, and a single-cycle router microarchitecture. FANIN and FANOUT together reduce full-system runtime for HyperTransport, a broadcast and ACK-intensive coherence protocol, by 19%, with up to 93.5% of the ACKs getting aggregated.

Together, SWIFT+FANOUT+FANIN is a highly optimized single-cycle per-hop NoC that can handle 1-to-1, 1-to-Many and Many-to-1 flows. However, the network latency in this design is always proportional to the number of hops that the flit wishes to traverse. As we continue to scale core counts, the average number of hops continues to increase as well leading to a corresponding increase in network latency. To counter this, we decoupled routers from links. We replaced clocked tri-state drivers within every crossbar by asynchronous/clock-less repeaters (i.e., inverters or a pair of inverters). This allows flits to bypass latches at every router and perform multi-hop traversals within a single-cycle before they are latched at their destination. We propose SMART: a router microarchitecture and flow control technique that lets flits dynamically setup and traverse multi-hop paths; the setup and traversal both take one-cycle each. In Appendix A we perform a design space exploration of repeated wires - varying wire spacing, repeater spacing, and repeater sizes. We observe that repeated wires can go 13-19 mm within 1ns (i.e., 1GHz clock frequency). For tile sizes of 1mm, this translates to 13-19 hops per cycle (HPC_{max}). For a network data-path with crossbars and repeaters at every hop, HPC_{max} drops to 11. It drops to 13 (9) along the network control-path through which the multi-hop straight (turning) path setup takes place. In the best case, if there is no contention, SMART allows flits to traverse multi-hop paths, all the way from the source up to HPC_{max} hops, within a cycle. If there is contention for a link, however, flits need to stop at the router connected to this link and arbitrate for it, just like in the baseline, making the achieved HPC lower than HPC_{max}. SMART with turns and a HPC_{max} of 8 reduces low-load network latency to 2 cycles across a variety of synthetic traffic patterns, breaking the dependence on number of hops. With a full-state directory protocol, SMART reduces full-system runtime by 27% and 52% respectively, across multiple benchmarks, for Private and Shared L2 implementations respectively. SMART, enhanced with FANOUT, speeds up broadcast delivery by 84% compared to FANOUT alone. SMART+FANOUT+FANIN reduces runtime by 15% and 19%, on average, across benchmarks running on HyperTransport and Token Coherence respectively.

The techniques presented in this thesis update the network latency (T_N) equation 1.1 for flits presented earlier in Chapter 1 to

$$T_N = \lceil (H/HPC) \rceil \cdot 2 + \sum_{h=1}^{H} t_c(h) \tag{7.1}$$

where H is the number of hops, HPC is the realized hops per cycle via SMART (equal to HPC_{max} in case of no contention), and $\sum_{h=1}^{H} t_c(h)$ is the overall contention delay along the route. The realized HPC depends on the number of routers at which there is contention. In the worst case of $t_c(h) > 0$ at every hop $h \in [1, H]$, the realized HPC will be 1, which is the same as the baseline. But if there is no contention for all the links that the flit wishes to traverse (i.e., $\forall h, t_c(h) = 0$), and H is less than HPC_{max}, the network latency is just 2 cycles.

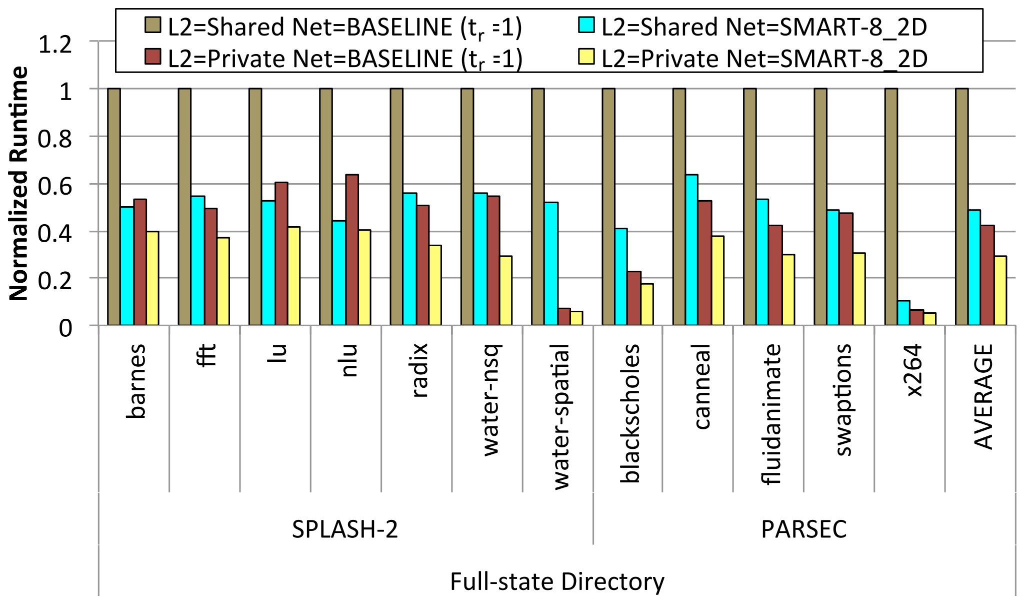


Figure 7-2: Private vs. Shared L2 Caches with BASELINE (t_r=1) and SMART.

7.2 Future Directions

The design of the memory subsystem within a multicore chip is a rich topic of research. Network latency plays a key role in the design process, since the on-chip traversal is often on the critical path of a memory request. We have shown that on-chip networks can be designed to be extremely low-latency (breaking the barrier between local and remote caches) and efficiently handle bursts of high-bandwidth traffic (such as multicasts and acknowledgements) at unicast latencies. These observations challenge the conventional wisdom that argues to keep data local and minimize communication. With the help of 2 case studies, we hope this thesis initiates a debate on the design of scalable cache hierarchies and protocols.

7.2.1 Case Study 1: Private vs. Shared L2

Coherence protocol designers often prefer Private L2s over Shared L2s to lower average cache access latencies, even though Shared L2 designs provide larger cache capacity. This is reflected in Figure 7-2 where a Private L2 is 57.4% faster on average than a Shared L2 with a baseline (t_r=1) network. However, a SMART network can turn this argument around its head. Shared L2s no longer need tens of cycles of access

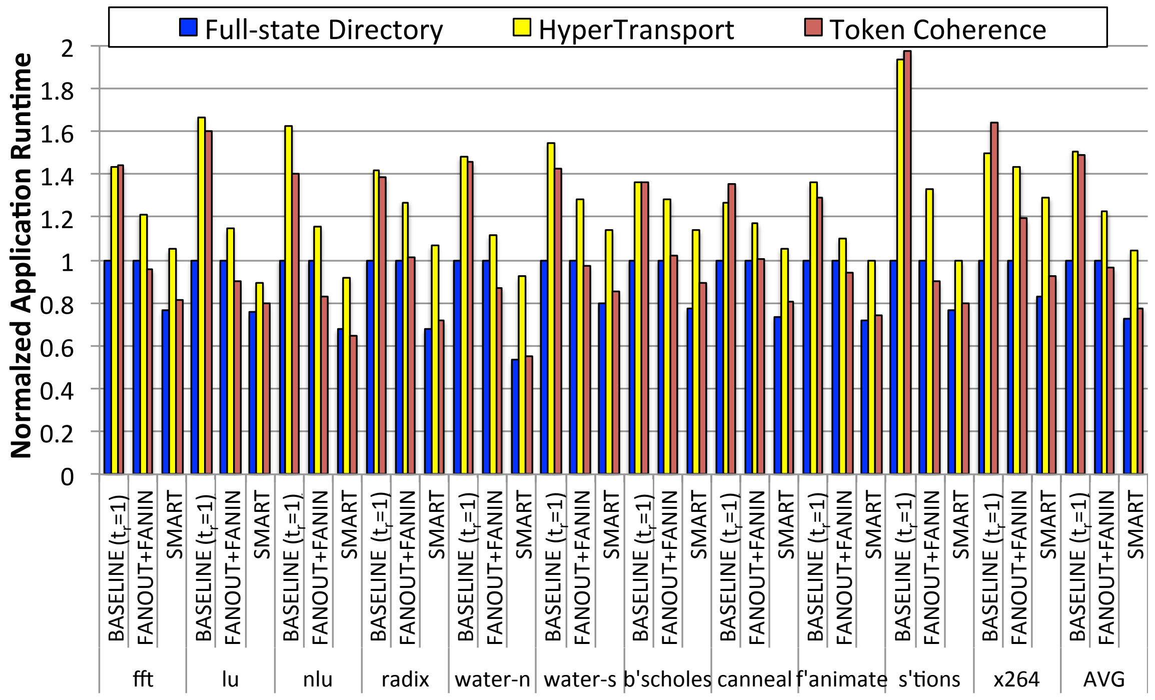


Figure 7-3: Directory vs. Broadcast protocols with BASELINE (t_r=1) and SMART.

latency. For some benchmarks, a Shared L2 with SMART performs as good (fft, water-nsq, swaptions, x264), and even better (barnes, lu, nlu) than a Private L2 with a baseline network. When both designs use SMART, Private is 40% faster on average. We believe that a coherence protocol and cache hierarchy that is aware that every HPC_{max} neighborhood is *local* can greatly benefit from SMART. The design of such a system would be a highly interesting and important research direction to pursue.

7.2.2 Case Study 2: Directory vs. Broadcast Protocols

The cache coherence protocol design space is fundamentally a trade-off between communication and storage. Broadcast protocols do not track the state of all on-chip lines, minimizing storage, at the cost of frequent 1-to-Many or 1-to-All messages. Directory protocols that track the state of all on-chip lines can send precise unicasts, minimizing communication, at the cost of directory storage structures which add indirection latency and area overheads.

In Figure 7-3 we plot the runtime of SPLASH-2 and PARSEC benchmarks with all 3 protocols used in this thesis (Full-state directory, HyperTransport and Token Coherence) with the BASELINE (t_r=1), FANOUT+FANIN, and SMART NoCs. For each benchmark, we normalize all runtimes to the runtime of the Full-state directory with the BASELINE (t_r=1) network. This allows us to compare the runtimes across both dimensions: network and protocol.

The runtime reductions of each individual protocol with the different network designs has been thoroughly explored in this thesis; so we focus on the other dimension[1]. With the BASELINE (t_r=1) network, which represents the best possible NoC design today, the full-state directory, which has the least network traffic, offers the lowest runtime - 48-51% lower on average - across all three protocols. This is in line with the conventional wisdom of network delay and throughput of broadcast protocols limiting performance. With the FANOUT+FANIN NoC, Token Coherence has comparable performance to the directory. HyperTransport is about 23% slower. The SMART NoC is able to lower the runtime of HyperTransport to be comparable to the runtime of a full-state directory running on the baseline NoC. These results point to a co-design of the coherence protocol and on-chip network as a potential research direction that stems directly from this thesis. An optimal design should sweep the communication vs. storage requirements of the protocol and weigh the latency, area, and power overheads at each point. The coherence protocol + network that meets the performance specifications within the power budget can then be picked.

7.2.3 Locality-Oblivious Shared Memory Design

Going forward, exposing a SMART-like network to the software layer can potentially open up multiple avenues for optimization. Applications with deterministic communication patterns, such as those in the SoC or GPU domain, could be mapped on the SMART NoC and paths preset for the entire duration of the application to pro-

[1] A direct comparison between the protocols using the runtime numbers from Figure 7-3 is not completely fair. This is because the full-state directory has higher storage requirements than both HyperTransport and Token Coherence; so the latter two should be compensated with larger caches. However, a design-space exploration of cache sizes is beyond the scope of this thesis and does not dilute the overall point that this case study is trying to make.

vide guaranteed single-cycle traversals unlike the current opportunistic design. This might be especially useful in a heterogeneous multicore, where certain tasks are tied to certain cores and cannot be moved to nearby cores to reduce the number of hops. Even in homogeneous multicores, data placement and movement is not a trivial task for the OS and compiler; and leveraging a SMART NoC could ease the burden on software developers from always optimizing for locality.

7.2.4 Scalability

As we move to thousands of cores on a chip in the exascale era, we need a scalable on-chip fabric to continue supporting shared memory systems. This thesis lays the ground work for such a fabric. As technologies scale, global wire delay is not expected to go down, as discussed in Appendix A. This is actually contrary to the trend for transistors, which become faster at smaller technology nodes. However, this is not really a problem, because our chip dimensions are expected to remain the same due to limitations of yields, and our clock frequencies are expected to remain the same because of the power wall. These three observations indicate that it will continue to be feasible to go from one end of the chip to the other in 1-2 cycles at 1GHz, even at future technology nodes. From an architectural perspective, smaller transistors will mean smaller tiles, and more hops. If we continue to design single-cycle per-hop designs, on-chip latency will be horrendous. But in a SMART design, smaller tiles will translate to a larger HPC_{max}, magnifying the benefits of SMART. We believe that a SMART-like design will not just be feasible, but also be required in multicore chips at future technology nodes.

7.2.5 Conclusion

As computing systems continue to pervade human society, computer architects need to ensure that our chips meet the performance and power requirements of all domains: embedded, graphics, high-performance, cloud, etc. Distributed computing (both on-chip and off-chip) relies on the network fabric for both efficiency and scalability. This thesis presented techniques for reducing on-chip latency to scale shared memory on-chip systems; but the ideas and insights presented are more general and can be applied to network fabrics in a variety of domains. We hope that the readers of this thesis shared the enthusiasm that went into the development of this text.

A Single-cycle Multi-hop Repeated Wires

Until you spread your wings, you'll have no idea how far you can fly.
- Anonymous

In this appendix, we perform a technology exploration of the wire delay of on-chip electrical wires. The goal of this study is to understand what design parameters can enable wires to send signals multiple hops within a target clock cycle. These single-cycle multi-hop wires act as motivation to design Single-cycle Multi-hop NoCs which were presented in Chapter 6.

A.1 Modeling and Evaluation Methodology

For all results in this appendix, we use the following modeling and evaluation methodology. We use the 45nm IBM SOI technology. We feed the parameters of the desired wire model (wire width (w), wire spacing (s), and repeater spacing (l)) to our automated wire layout tool [19][1] to create a place-and-routed 128-bit wire layout, as shown in Figure A-1. Next, we perform Design Rule Check (DRC), Layout vs. Schematic (LVS) and Parasitic Extraction (PEX) of the design using Cadence Encounter. We feed the extracted spice netlist generated by PEX through Cadence Ultrasim and perform circuit simulations in mixed-signal mode to get delay and energy numbers. The testbench connects the extracted wire between two flip flops, feeds random traffic,

[1]The automated layout tool was created by Owen Chen.

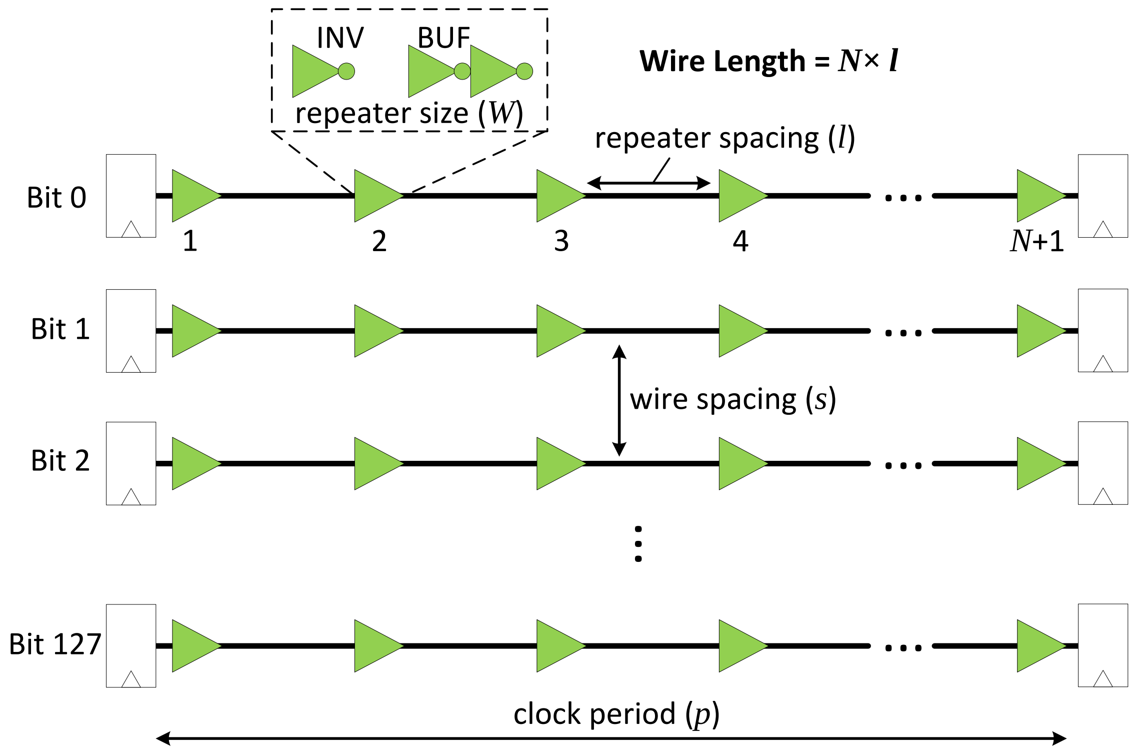


Figure A-1: Experimental setup to measure delay of repeated wires.

and compares the input and output values. The experiment involves increasing the length of the wire till it fails timing (i.e., the output and input no longer match) at the desired clock period (p) (1ns unless specified). We sweep through 8 repeater sizes (W) - 1X, 3X, 5X, 7.5X, 9X, 11X, 13X and 16X - and 2 repeater models - inverter and buffer. For each data point (i.e., wire length), we plot the minimum energy point across all the 16 repeater circuits which meet timing. All graphs plot the energy in fJ/bit/mm and the length in mm. We use close to the minimum wire width on metal layer M6 in all experiments.

A.2 Wire Delay

A wire can be modeled as a distributed RC network driven by a driver (single/multi-stage inverter), as shown in Figure A-2. A simple equation for the delay through this wire is [36][2]:

[2]This model ignores both non-linear drive resistance as well as the effect of slew rate on delay.

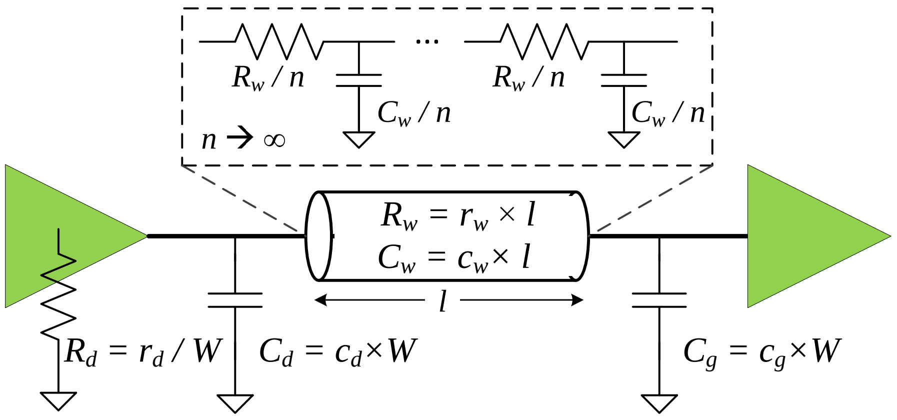


Figure A-2: Model of Wire + Driver/Repeater.

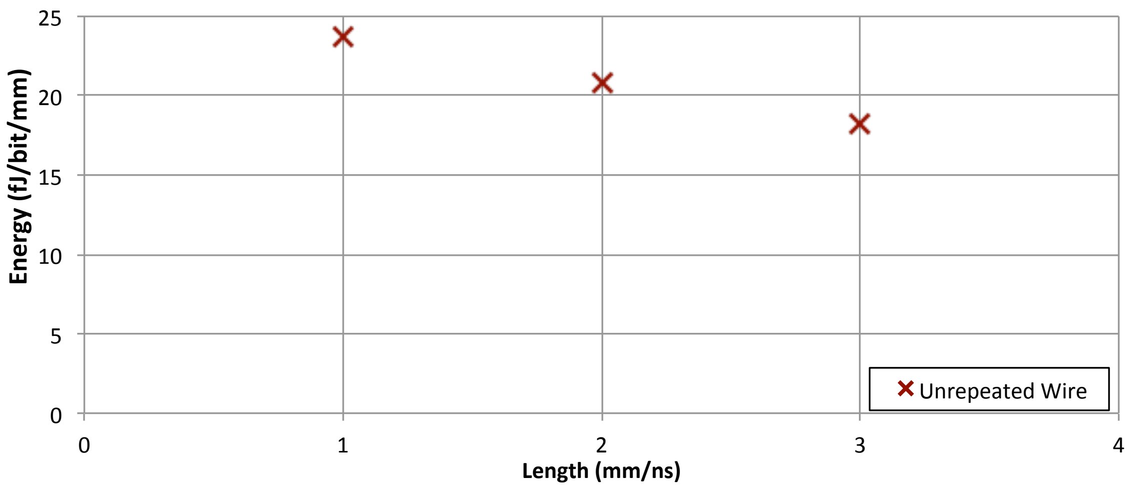


Figure A-3: Delay of unrepeated wire. ($s \sim 3 \cdot DRC_{min}$, $p =$ 1ns)

$$\begin{aligned} Delay &= 0.69[D_{driver} + D_{wire}] \\ &= 0.69[R_d C_d + (R_d(C_w + C_g) + R_w(\frac{1}{2}C_w + C_g))] \end{aligned} \tag{A.1}$$

In Figure A-3 we plot the energy vs. length for an unrepeated wire. We can see that the unrepeated wire can transmit signals up to 3mm before it fails timing.

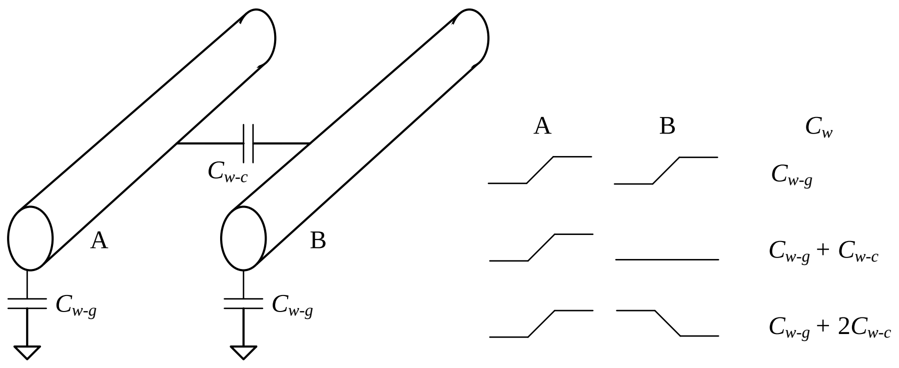


Figure A-4: Capacitive Coupling.

A.3 Repeated Wires

Adding repeaters, i.e., inverters or buffers (pair of inverters), at regular intervals along a long wire is a standard technique for reducing wire delay [70, 45]. The intuition is that the RC delay goes up as square of the wire's length (since both R and C go up linearly with the wire's length). Breaking the long wire into multiple stages makes the delay go up linearly with the number of stages instead. The equation for the delay through a repeated wire is [36]:

$$\begin{aligned} Delay &= N \cdot D_{stage} \\ &= N \cdot 0.69[D_{repeater} + D_{wire}] \\ &= N \cdot 0.69[R_d C_d + (R_d(C_w + C_g) + R_w(\frac{1}{2}C_w + C_g))] \\ &= N \cdot 0.69[\frac{r_d}{W} c_d W + (\frac{r_d}{W}(c_w l + c_g W) + r_w l(\frac{1}{2}c_w l + c_g W))] \end{aligned} \tag{A.2}$$

A.3.1 Impact of wire spacing

Wire spacing s affects the coupling capacitance (C_{w-c}) between adjacent wires. Coupling capacitance can increase the effective capacitance of the wire (C_w) depending on the value being transmitted on the wires. Figure A-4 shows this effect for two wires. If both wires transmit the same value (0 or 1), the effective capacitance is C_{w-g} (i.e., capacitance to ground). If one wire does not switch while the other switches, the

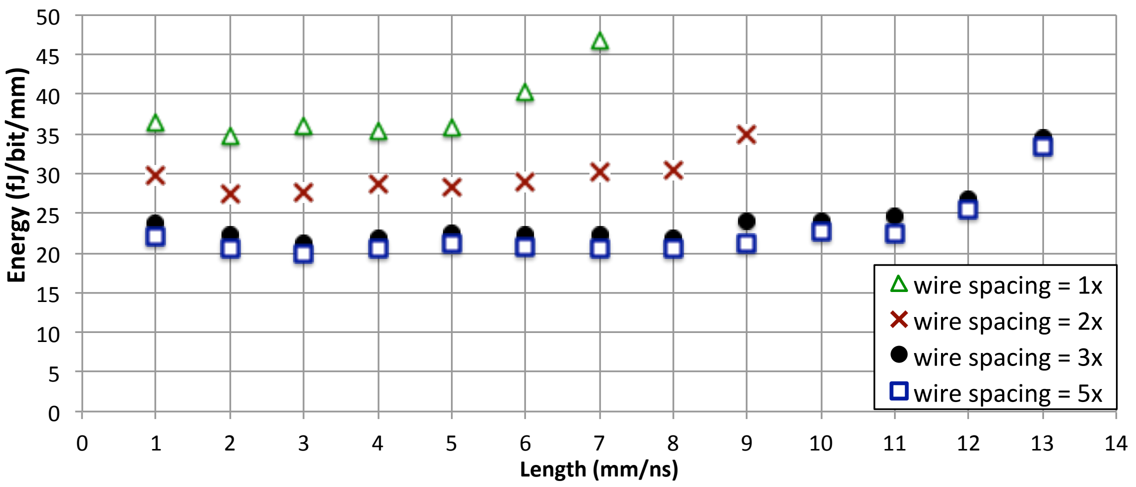


Figure A-5: Impact of wire spacing (s). (l = 1mm, p = 1ns)

effective capacitance is $C_{w-g} + C_{w-c}$. If both wires switch in opposite directions, the effective capacitance is $C_{w-g} + 2C_{w-c}$. For a parallel set of three or more wires this effect gets magnified. If a wire switches in the opposite direction to both its neighbors, its effective capacitance can go up to $C_{w-g} + 4C_{w-c}$.

Increased wire spacing lowers the coupling capacitance C_{w-c} between wires, thereby increasing its speed. Figure A-5 plots the measured energy vs. length graph for a repeated wire with varying wire spacing (in multiples of a value close to DRC$_{min}$[3]). We can see that with 1X wire spacing, the wire transmits up to 7mm and consumes 47 fJ/bit/mm which is 2.26X higher than the energy consumed by the 5X spacing design at 7mm. With 2X spacing, the lowered capacitance allows the transmission distance to go up to 9mm, at 35 fJ/bit/mm. With 3X spacing, the maximum distance goes up to 13mm at 35 fJ/bit/mm. Increasing spacing beyond 3X does not help in delay and lowers energy very slightly. Thus we use 3X spacing in all the following experiments.

A.3.2 Impact of repeater spacing

Repeater spacing l affects the number of stages and therefore the delay, as shown in Equation A.2. Theoretically, solving the equation to minimize delay gives l =

[3]We cannot report specific values of our wire width and spacing due to process design kit confidentiality issues.

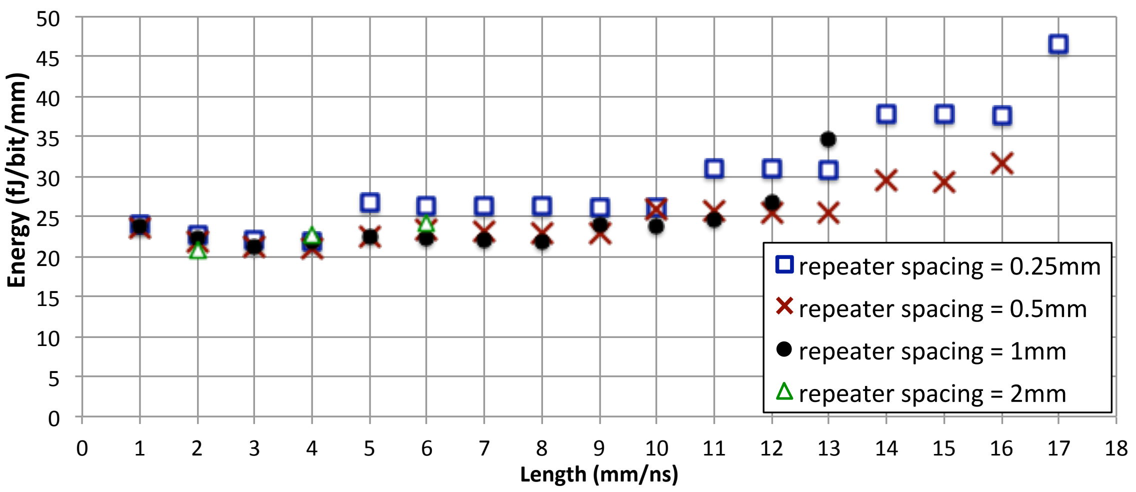


Figure A-6: Impact of repeater spacing (l). ($s \sim 3 \cdot DRC_{min}$, $p = 1$ns)

0.4mm. Figure A-6 plots the measured energy vs. length graph for a repeated wire with varying repeater spacing. We can see that repeaters at 0.5 mm and 0.25mm allow the wire to transmit up to 16mm and 17mm respectively. Repeaters at 1mm spacing decrease the maximum transmission length to 13mm. At a spacing of 2mm, the repeated wire can only transmit up to 6mm.

It is not always possible to layout repeaters at the spacing that minimizes delay due to constraints related to layout of other components of the design. For instance, in multicore chips the tile size (assumed to be 1mm in this thesis) determines the repeater spacing since we can place the repeaters (logic) at the crosspoints of the router's crossbar. The wires themselves can be routed on higher metal layers over the core and router logic.

A.3.3 Impact of repeater size

Figure A-7 plots the energy vs. length graph for a repeated wires with varying driver (repeater) sizes. At every data point, we pick the minimum energy point between the inverter and the buffer. The minimum sized driver can drive the wire up to 4mm, at the lowest energy among all other drivers. The 3X and 5X drivers push the maximum length to 9mm and 13mm respectively, at the cost of 5% and 17% higher energy/bit/mm than the minimum sized driver. Drivers of sizes of 7.5X -

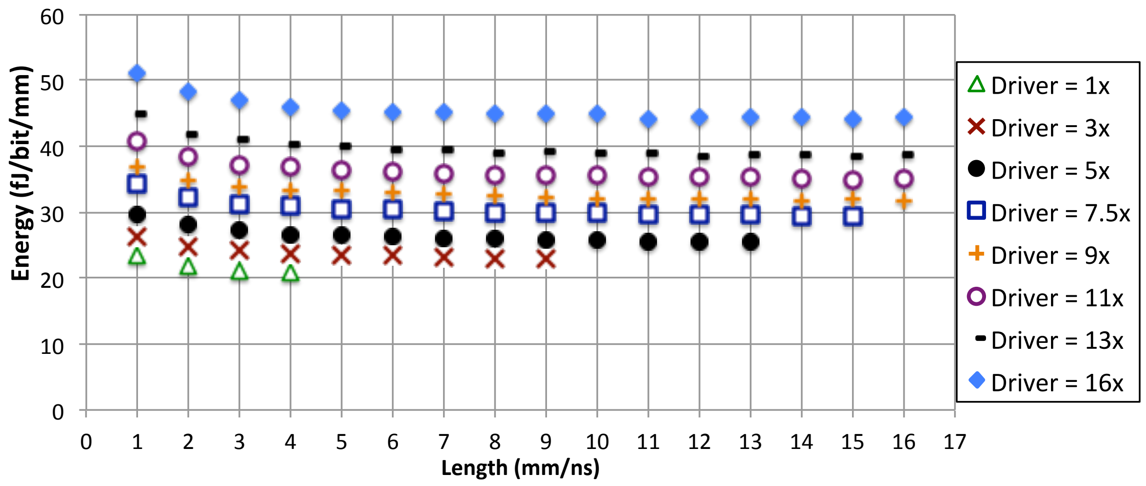


Figure A-7: Impact of repeater size (W). ($s \sim 3 \cdot DRC_{min}$, $l = 0.5mm$, $p = 1$ns)

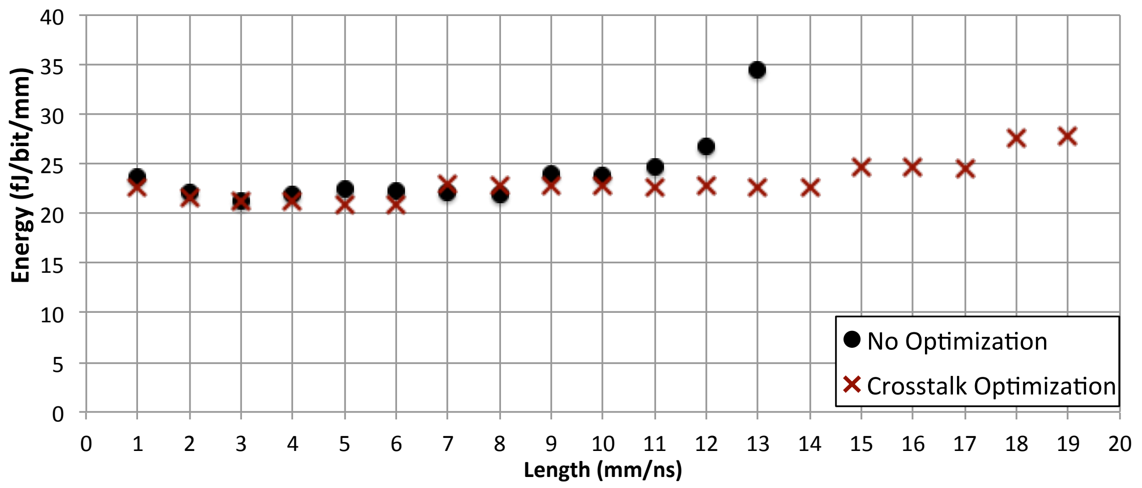


Figure A-8: Impact of crosstalk. ($s \sim 3 \cdot DRC_{min}$, $l = 1mm$, $p = 1$ns)

16X are required to push the design further to 15-16mm at the cost of 15% to 73% higher energy/bit/mm compared to the 5X driver. The theoretical driver size W for minimum delay[4] is 20X but could not be modeled because the largest standard cell in the technology library was for the 16X driver.

A.3.4 Impact of crosstalk

The coupling capacitance between adjacent wires results in crosstalk, which affects the maximum length that can be traversed by the wire without bit errors, as discussed

[4]Obtained by partially differentiating Equation A.2 with respect to W.

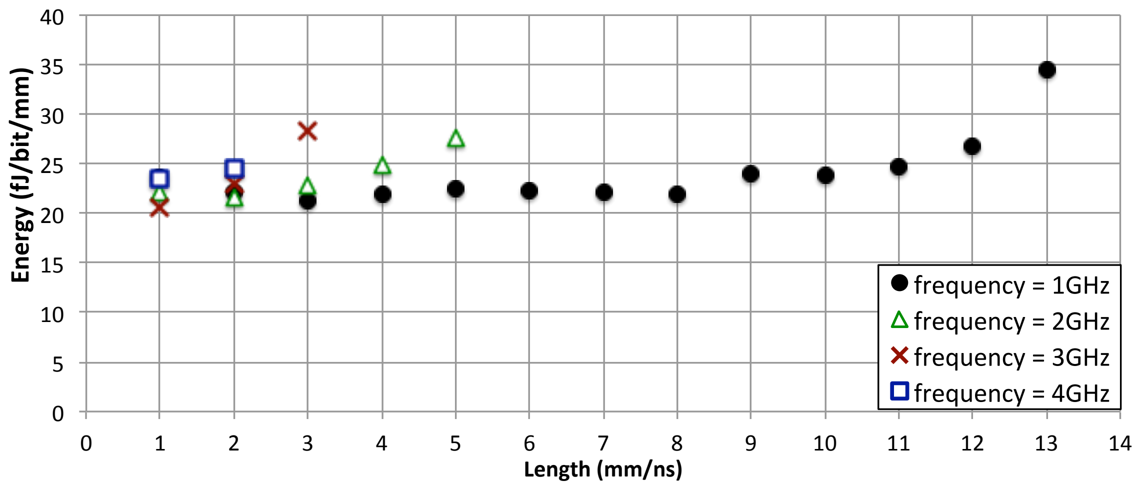


Figure A-9: Impact of clock frequency. ($s \sim \mathbf{3} \cdot DRC_{min}$, $l = \mathbf{1}mm$)

earlier in Section A.3.1.. We enabled an optimization in Cadence Encounter during parasitic extraction that removes the coupling capacitor between adjacent wires and instead models it as additional grounded capacitors on the adjacent wires. This approximation does not affect the energy calculations much, but affects the delay estimates since it does not fully model crosstalk effects. With this optimization enabled, Figure A-8 shows that the repeated wire with 1mm repeater spacing is able to go up to 19mm, as compared to 13mm in the original design.

This design represents the lowest possible delay for the repeated wire if crosstalk is mitigated. A real design that uses circuit tricks like crossover [70] and shielding wires [70] to lower crosstalk should be able to push the repeated wire to transmit up to 13mm-19mm.

A.3.5 Impact of frequency

Figure A-9 plots the energy vs. length graph for a repeated wire at varying clock frequencies. At 2GHz and 3GHz, the maximum length that the wire can be driven drops down to 5mm and 3mm respectively. This is less than half and one-third of 13mm (the maximum length at 1GHz). The reason for the super-linear slowdown is the slew rate of the signal. The signal has some rise time and fall time delay. At higher frequencies (i.e., shorter clock periods), the repeater is unable to produce

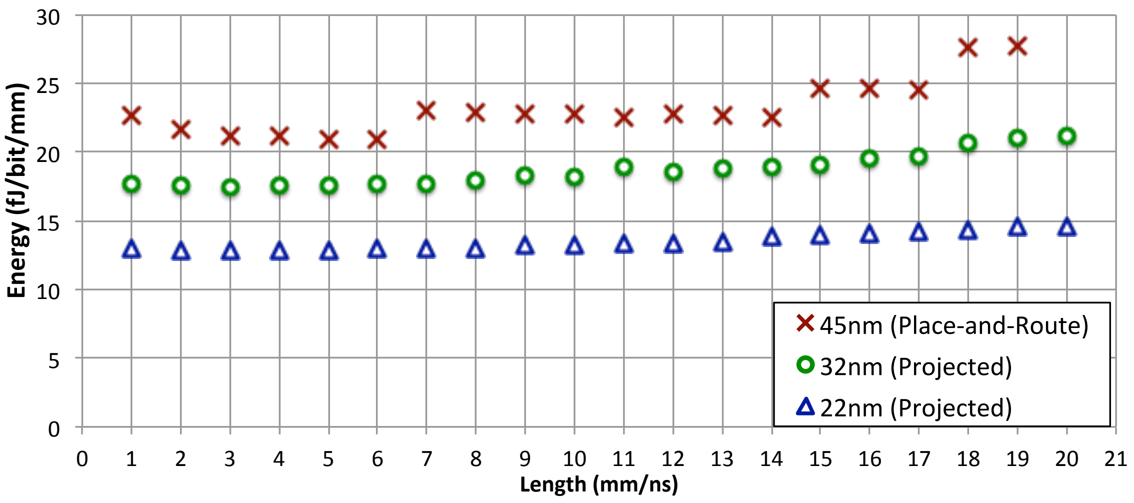


Figure A-10: Impact of technology scaling. ($s \sim 3 \cdot DRC_{min}$, $l = 1mm$, $p = 1$ns)

the extra current required to reduce the rise/fall time proportionally, which ends up becoming a larger percentage of the wire delay. This could be mitigated by using an even larger driver at high frequencies. At 4GHz the signal can be driven up to 2mm, and beyond that it cannot be driven beyond 1mm.

A.3.6 Impact of technology

Figure A-10 plots the energy vs. length graph for repeated wires at 45nm, 32nm and 22nm technology nodes. The 32nm and 22nm numbers are projections using DSENT [76] - a timing-driven power modeling tool for on-chip networks. The 45nm values are from place-and-route, using the crosstalk optimization discussed earlier in Section A.3.4, since DSENT does a similar optimization. At 45nm, the repeated wire drives signals up to 19mm. At 32nm and 22nm, the repeated wire drives signals upto 20mm at 20% and 42% lower energy/bit/mm on average respectively.

With technology scaling, the delay of repeated wires is expected to remain similar [36, 20]. The reason for this trend is as follows. As wire widths become thinner, their resistance R_w goes up. To compensate for this increase, the height of the (higher) metal layers has been continuously increasing to keep R_w almost constant. But at lower technology nodes, the height cannot be increased further [2] resulting in R_w going up. However, this rise in R_w gets compensated by a lower capacitance to ground

C_{w-g} in the thinner wires. If wires are placed at their minimum spacing, coupling capacitance C_{w-c} will go up. But if we give 3X or more spacing, as we showed in Figure A-5, the effects of coupling capacitance can be reduced significantly. Thus the overall delay of the wire ($R_w \times C_{w-g}$) is expected to remain fairly constant as technology scales. Transistors, on the other hand, become faster at lower technologies due to lower capacitances and only slightly higher resistances. Transistors becoming faster compared to wires has been projected in the past as a motivation to keep global chip-wide communication to a minimum and heavily optimize for locality [47, 35]. However, we believe that this trend does not in fact hurt communications. Even if technology scales, chip dimensions are expected to remain fairly constant (due to yields) and clock frequencies have plateaued off (due to the power wall described in Chapter 1). Thus the *number of cycles* it takes to go from a core at one end of the chip to the other is expected to remain the same, and not go down. This motivates Chapter 6 where we leverage repeated wires on the data-path to perform single-cycle traversals across the chip.

Bibliography

[1] Intel Nehalem. http://www.realworldtech.com/nehalem.

[2] International Technology Roadmap for Semiconductors. http://www.itrs.net.

[3] The Angstrom Project. http://projects.csail.mit.edu/angstrom.

[4] Wind River Simics Full System Simulation. http://www.windriver.com/products/simics.

[5] J. L. Abellán, J. Fernández, and M. E. Acacio. Efficient and Scalable Barrier Synchronization for Many-Core CMPs. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, pages 73–74, 2010.

[6] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[7] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A Detailed On-chip Network Model inside a Full-system Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, 2009.

[8] N. Agarwal, L.-S. Peh, and N. K. Jha. In-Network Snoop Ordering (INSO): Snoopy Coherence on Unordered Interconnects. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 67–78, 2009.

[9] A. Ahmed, P. Conway, B. Hughes, and F. Weber. AMD Opteron Shared Memory MP Systems. In *In Proceedings of the 14th HotChips Symposium*, 2002.

[10] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-Threaded Workloads. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 7–18, 2003.

[11] J. Bae, J.-Y. Kim, and H.-J. Yoo. A 0.6pJ/b 3Gb/s/ch Transceiver in 0.18um CMOS for 10mm On-chip Interconnects. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2861–2864, 2008.

[12] J. Bainbridge and S. Furber. Chain: A Delay-Insensitive Chip Area Interconnect. *IEEE Micro*, 22(5):16–23, Sept 2002.

[13] J. Balfour and W. J. Dally. Design Tradeoffs for Tiled CMP On-Chip Networks. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 187–198, 2006.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[15] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 294–304, 1999.

[16] N. Binkert *et al.* The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[17] T. Bjerregaard and J. Sparso. A Router Architecture for Connection-Oriented Service Guarantees in the MANGO Clockless Network-on-Chip. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, pages 1226–1231, 2005.

[18] J. G. Castanos, L. Ceze, K. Strauss, and H. S. Warren Jr. Evaluation of a Multithreaded Architecture for Cellular Computing. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 311–322, 2002.

[19] C.-H. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan, and L.-S. Peh. SMART: A Single-Cycle Reconfigurable NoC for SoC Applications. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, pages 338–343, 2013.

[20] G. Chen, H. Chen, M. Haurylau, N. Nelson, D. Albonesi, P. M. Fauchet, and E. G. Friedman. Electrical and Optical On-Chip Interconnects in Scaled Microprocessors. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2514–2517, 2005.

[21] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, Mar 2010.

[22] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 137–150, 2004.

[24] 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. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[25] J. Duato, S. Yalamanchili, and L. M. Ni. *Interconnection Networks: An Engineering Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[26] P. A. Fidalgo, V. Puente, and J.-Á. Gregorio. MRR: Enabling Fully Adaptive Multicast Routing for CMP Interconnection Networks. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 355–366, 2009.

[27] M. Galles. Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER Chip. In *Proceedings of IEEE Annual Symposium on High Performance Interconnects (HOTI)*, pages 141–146, 1996.

[28] A. Gara *et al.* Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development.*, 49(2):195–212, Mar 2005.

[29] C. J. Glass and L. M. Ni. The Turn Model for Adaptive Routing. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 278–287, 1992.

[30] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Compututers*, 32(2):175–189, Feb 1983.

[31] P. Gratz, C. Kim, K. Sankaralingam, H. Hanson, P. Shivakumar, S. W. Keckler, and D. Burger. On-Chip Interconnection Networks of the TRIPS Chip. *IEEE Micro*, 27(5):41–50, Sept 2007.

[32] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Express Cube Topologies for On-Chip Interconnects. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 163–174, 2009.

[33] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 401–412, 2011.

[34] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *In Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 312–321, 1990.

[35] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 184–195, 2009.

[36] Ron Ho. *On-Chip Wires: Scaling and Efficiency.* PhD thesis, Stanford University, Aug 2003.

[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 2007.

[38] J. Howard *et al.* A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 108–109, 2010.

[39] T. N. K. Jain, P. V. Gratz, A. Sprintson, and G. Choi. Asynchronous Bypass Channels: Improving Performance for Multi-synchronous NoCs. In *Proceedings of the IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 51–58, 2010.

[40] N. D. E. Jerger and L.-S. Peh. *On-Chip Networks.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[41] N. D. E. Jerger, L.-S. Peh, and M. Lipasti. Virtual Circuit Tree Multicasting: A Case for On-chip Hardware Multicast Support. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 229–240, 2008.

[42] D. Johnson *et al.* Rigel: A 1,024-Core Single-Chip Accelerator Architecture. *IEEE Micro*, 31(4):30–41, Jul 2011.

[43] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, pages 423–428, 2009.

[44] Y.-H. Kao, M. Yang, N. S. Artan, and H. J. Chao. CNoC: High-Radix Clos Network-on-Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(12):1897–1910, 2011.

[45] B. Kim and V. Stojanović. Equalized Interconnects for On-Chip Networks: Modeling and Optimization Framework. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 552–559, 2007.

[46] B. Kim and V. Stojanovic. A 4Gb/s/ch 356fJ/b 10mm Equalized On-Chip Interconnect with Nonlinear Charge-Injecting Transmit Filter and Transimpedance Receiver in 90nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 66–67, 2009.

[47] C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 211–222, 2002.

[48] J. Kim, J. Balfour, and W. J. Dally. Flattened Butterfly Topology for On-Chip Networks. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 172–182, 2007.

[49] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta. Microarchitecture of a High-Radix Router. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 420–431, 2005.

[50] T. Krishna, C.-H. O. Chen, W.-C. Kwon, and L.-S. Peh. Breaking the On-Chip Latency Barrier Using SMART. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 378–389, 2013.

[51] T. Krishna, A. Kumar, L.-S. Peh, J. Postman, P. Chiang, and M. Erez. Express Virtual Channels with Capacitively Driven Global Links. *IEEE Micro*, 29(4):48–61, 2009.

[52] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt. Towards the Ideal On-Chip Fabric for 1-to-Many and Many-to-1 Communication. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 71–82, 2011.

[53] T. Krishna, J. Postman, C. Edmonds, L.-S. Peh, and P. Chiang. SWIFT: A SWing-reduced Interconnect For a Token-based Network-on-Chip in 90nm CMOS. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 439–446, 2010.

[54] V. Krishnan and J. Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. *International Journal of Parallel Programming*, 29(1):3–33, Feb 2001.

[55] A. Kumar, P. Kundu, A. P. Singh, L.-S. Peh, and N. K. Jha. A 4.6Tbits/s 3.6GHz Single-cycle NoC Router with a Novel Switch Allocator in 65nm CMOS. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 63–70, 2007.

[56] A. Kumar, L.-S. Peh, and N. K. Jha. Token Flow Control. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 342–353, 2008.

[57] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha. Express Virtual Channels: Towards the Ideal Interconnection Fabric. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 150–161, 2007.

[58] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *Proceedings of the IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 477–488, 2010.

[59] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 241–251, 1997.

[60] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 148–159, 1990.

[61] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 182–193, 2003.

[62] M. M. K. Martin *et al.* Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, (4):92–99, 2005.

[63] H. Matsutani, M. Koibuchi, H. Amano, and T. Yoshinaga. Prediction Router: Yet Another Low Latency On-Chip Router Architecture. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 367–378, 2009.

[64] G. Michelogiannakis, J. D. Balfour, and W. J. Dally. Elastic-Buffer Flow Control for On-Chip Networks. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 151–162, 2009.

[65] R. Mullins, A. West, and S. Moore. Low-Latency Virtual-Channel Routers for On-Chip Networks. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 188–197, 2004.

[66] S. Park, T. Krishna, C.-H. O. Chen, B. Daya, A. P. Chandrakasan, and L.-S. Peh. Approaching the Theoretical Limits of a Mesh NoC with a 16-Node Chip Prototype in 45nm SOI. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 398–405, 2012.

[67] L.-S. Peh and W. J. Dally. A Delay Model and Speculative Architecture for Pipelined Routers. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 255–266, 2001.

[68] G. F. Pfister *et al.* The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *In Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 764–771, 1985.

[69] J. Postman, T. Krishna, C. Edmonds, L.-S. Peh, and P. Chiang. SWIFT: A Low-Power Network-On-Chip Implementing the Token Flow Control Router Architecture With Swing-Reduced Interconnects. *IEEE Transactions on VLSI Systems (TVLSI)*, 21(8):1432–1446, 2013.

[70] J. Rabaey and A. Chandrakasan. *Digital Integrated Circuits: A Design Perspective.* Prentice Hall Pub., 2002.

[71] A. Raghavan, C. Blundell, and M. M. K. Martin. Token tenure: PATCHing token counting using directory-based cache coherence. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 47–58, 2008.

[72] S. Rodrigo, J. Flich, J. Duato, and M. Hummel. Efficient Unicast and Multicast Support for CMPs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 364–375, 2008.

[73] F. A. Samman, T. Hollstein, and M. Glesner. Multicast Parallel Pipeline Router Architecture for Network-on-Chip. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, pages 1396–1401, 2008.

[74] S. Scott, D. Abts, J. Kim, and W. J. Dally. The BlackWidow High-Radix Clos Network. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 16–28, 2006.

[75] K. Strauss, X. Shen, and J. Torrellas. Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 327–342, 2007.

[76] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *Proceedings of the IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 201–210, 2012.

[77] M. B. Taylor *et al.* The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, Mar 2002.

[78] H. Wang, L.-S. Peh, and S. Malik. Power-driven Design of Router Microarchitectures in On-chip Networks. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–116, 2003.

[79] L. Wang, Y. Jin, H. Kim, and E. J. Kim. Recursive Partitioning Multicast: A Bandwidth-Efficient Routing for Networks-on-Chip. In *Proceedings of the IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 64–73, 2009.

[80] M. A. Watkins and D. H. Albonesi. ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 497–508, 2010.

[81] D. Wentzlaff *et al.* On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, Sept 2007.

[82] 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 IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 24–36, 1995.