UNIVERSITY OF CALIFORNIA, SAN DIEGO

Making the On-Chip World Smaller with Low-Latency On-Chip Networks

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Yashar Asgarieh

Committee in charge:

Professor Bill Lin, Chair
Professor Chung-Kuan Cheng
Professor Ranjit Jhala
Professor Truong Nguyen
Professor Dean M. Tullsen

2017

The dissertation of Yashar Asgarieh is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2017

DEDICATION

To my mother, who has patiently supported me at every step of my life.

To my late father, who taught me the values by which I have lived my life.

To my family.

EPIGRAPH

To wisely live your life, you don't need to know much

Just remember two main rules for the beginning:

You better starve, than eat whatever

And better be alone, than with whoever.

—Omar Khayyám

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Prof. Bill Lin, for his support. I feel very grateful to have had the privilege of working with him, as an advisor and teacher. I would like to thank Prof. Dean Tullsen for his guidances and helps toward this dissertation. Next, I would like to thank my other dissertation committee members, Prof. Ranjit Jhala, Prof. C.K. Cheng, and Prof. Truong Nguyen for their advice and comments. I would also like to thank all my colleagues and friends at UCSD for making years in graduate school unforgettable. Especially, I would like to thank the department colleagues and soccer teammates Professor Yannis Papakonstantinou, Matt Der, Stefan Schneider, Chris Tosh, Michael Walter, with whom we earned the intramural game's championship. The other amazing folks that I owe a huge debt of gratitude are CSE staff members, especially Julie Conner for guiding me through administrative hurdles countless times during these years. My greatest thanks go to my family for offering me their love and support, and holding my back whenever I needed the most. There are many many other dear friends that I would like to thank all of them. Unfortunately, the list is very long and time is short. I wish all of them the best in their life from the bottom of my heart, and I would like to let them know their friendship is one of the most precious things that earned in these years which I hope I can carry it with myself for years and years yet to come.

Chapter 3, in part, is in part, is currently being prepared for submission for publication of the material. Asgarieh, Yashar; Lin, Bill. The dissertation author was the primary investigator and author of this material.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the

IEEE/ACM International Symposium on Networks-on-Chip (NOCS) 2016. Asgarieh, Yashar; Lin, Bill. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is in part, is currently being prepared for submission for publication of the material. Asgarieh, Yashar; Lin, Bill. The dissertation author was the primary investigator and author of this material.

VITA

2006	Bachelor of Science in Computer Engineering Iran University of Science and Technology
2009	Master of Science in Computer Engineering Sharif University of Technology
2017	Doctor of Philosophy in Computer Science (Computer Engineering) University of California, San Diego

PUBLICATIONS

H. Mahmoodi, S.S. Lakshmipuram, M. Arora, Y. Asgarieh, H. Homayoun, B. Lin, D.M. Tullsen, Resistive computation: A critique, IEEE Computer Architecture Letters, 13(2), pp.89-92. 2014.

Y. Asgarieh, B. Lin, Sharing a global on-chip transmission line medium without centralized scheduling, In Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS), pp. 1-8, August 2016.

Y. Asgarieh, C.K. Cheng, B. Lin, The design of global on-chip networks with repeated equalized transmission lines, ACM Transactions on Design Automation of Electronics Systems (TODAES), first round revision of submitted manuscript, 2016.

Y. Asgarieh, B. Lin, Quadratic reduction in SSR arbitration complexity with false negatives avoidance, ACM Transactions on Architecture and Code Optimization (TACO), to be submitted, 2017.

Y. Asgarieh, B. Lin, A fast and accurate NoC-centric simulator via instruction trace simulation, ACM Transactions on Design Automation of Electronics Systems (TODAES), to be submitted, 2017.

ABSTRACT OF THE DISSERTATION

Making the On-Chip World Smaller with Low-Latency On-Chip Networks

by

Yashar Asgarieh

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2017

Professor Bill Lin, Chair

Multi-core processors have rapidly grown in core count since the first commercial dual-core processor in 2001. Today, general-purpose multi-cores with 32 cores and embedded multi-cores with over 100 cores are available, with increasing core counts still to come. To enable multi-cores to run many different applications, the solution of choice has been to connect the cores by a shared Network-on-Chip (NoC) so that any communication pattern can be supported. However, previous NoC designs are not scalable in terms of network latency when the communicating cores are not nearby each other.

Unfortunately, high network latencies create performance bottlenecks and limit the flexible usage of on-chip resources. Computer architects have sought to avoid interactions between far away cores, but the effectiveness of locality optimizations are diminishing.

In this thesis, we propose to make the on-chip world appear *smaller* by providing extremely low-latency networks that can make far away resources appear much closer. This is achieved by leveraging specially-engineered electrical wires that can transport data across chip at both high data rates and low latencies. We first investigate the use of asynchronous repeated wires that run across a shared hop-by-hop 2D mesh network. Using these asynchronous repeated wires, we can configure routers to bypass their pipelines to create single-cycle paths across multiple routers. To allocate these single-cycle multi-hop paths, we present a novel arbitration scheme that has low implementation complexity, guarantees correctness, and avoids throughput loss. We also investigate the use of on-chip transmission lines that conduct signals at the speed of light at extremely high data rates. We present a shared medium architecture for global on-chip communications using these transmission lines, and we present several fully-distributed arbitration schemes for controlling access to this shared medium. In addition, we present a fast and accurate NoC simulation methodology that accounts for complex interactions between the NoC and the application, memory sub-system, and processing cores. This simulation approach can be used to effectively evaluate NoC designs, including those described in this thesis.

Chapter 1

Introduction

With the end of Dennard scaling [26], computer architects have turned to parallelism and multi-core architectures to continue performance improvements. Over the last decade, we have seen a continuous increase in the number of cores on a chip, as enabled by the corresponding scaling in CMOS technologies. For example, Intel and AMD have both recently announced 32-core x86 general-purpose server chips [3, 1]. For embedded applications, Mellanox has announced processors with 100 ARM cores [4][1]. For smartphone and tablet applications, heterogeneous MPSoCs (multi-processor systems-on-chips) with many specialized cores are widely deployed [6]. Finally, computer architects are also exploring accelerator-rich heterogeneous architectures with many specialized accelerators as well as general-purpose CPU cores as a way to continue performance growth while improving energy efficiency [28, 33, 23]. The push towards accelerator-rich architectures is largely driven by the prediction of *dark silicon* [28, 33] in which most parts of the silicon are expected to be off by default; in turn, specialized accelerators are only powered

[1]This 100-ARM-core chip is based on the Tilera GX-100 processor. Mellanox acquired Tilera through its acquisition of EZchip.

on when needed.

Central to the performance of multi-core processors is the design of the interconnection fabric that enables communications among cores. For scalability, a widely used solution is a Network-on-Chip (NoC). In particular, NoCs based on a two-dimensional mesh have been widely studied, as depicted in Fig. 2.1. Communications between cores are achieved by routing messages hop-by-hop from the source to the destination. Although mesh networks are scalable in size, their effectiveness at non-local communications quickly diminishes with increasing number of on-chip cores due to long on-chip latencies, which complicate the design of cache coherence protocols and can significantly diminish the effectiveness of caching, among other problems. Delays due to router pipelines, queuing, and serialization all contribute towards a much longer on-chip latency than an ideal point-to-point interconnect. Besides the on-chip latency problem, the traffic load on each link also increases with the network diameter, which diminishes the effective throughput. As shown in [44], long on-chip latencies have substantial negative impact on application performance.

Although researchers have sought to limit the impact of long on-chip latencies by avoiding long-distance data transfers, locality-aware approaches are becoming less and less effective. For example, substantial research on locality-aware designs have focused on keeping copies of data at private local caches. However, the use of private local caches is becoming more and more complicated with increasing network diameter due to the problems of data tracking and invalidation: using a directory or full-chip broadcasts to track the states of all on-chip caches are both extremely expensive and difficult to scale.

Besides the challenges associated with maintaining cache locality, the scheduling

of threads is also a difficult challenge with an increasing number of cores. For example, an operating system can improve transactions latencies by placing interacting threads closer to the shared data and to each other. However, the thread scheduling task is itself a complicated optimization problem which requires having information about the status of all existing threads and system resources [80]. Besides the status of the current threads, the operating system should take into account other system dynamics (e.g., temperature, power), which can lead to suboptimal solutions that are not locality-friendly. In multi-tenant cloud computing deployments, the scheduling of threads to maintain locality becomes considerably more difficult as newly spawned threads may have to be scheduled on to a highly fragmented pool of resources.

In addition, low-latency long-distance communications are essential for accelerator-rich architectures in the dark silicon era: the ensemble of accelerators needed for an application are unlikely to be near each other; but yet, they must act in concert to effectively perform a given task. Heterogeneous MPSoCs also need low-latency long-distance communications as more and more specialized cores are integrated.

Finally, besides low on-chip latency, all applications described above require high-throughput, flexibility, and scalability as well. For example, deep learning applications have both training and inference phases. Whereas the training phase is mostly throughput intensive, the inference phase is both latency and throughput intensive [24, 17]. Thus, an NoC design that is merely optimized for just throughput or just latency will not be a good match for many applications. Further, an NoC design has to be flexible in order to adapt to a variety of changing traffic patterns, depending on use cases. This precludes dedicated interconnects as viable solutions. Lastly, an NoC design has to be

scalable, including the scalability of any resource allocation mechanism. This objective favors distributed mechanisms that can scale with increasing network sizes.

1.1 Thesis Contributions and Organization

The primary contributions of this thesis include two NoC designs that can achieve extremely low on-chip latencies for long-distance communications. The first design is based on single-cycle multi-hop traversals using asynchronous repeated wires. In contrast to earlier works that use a similar repeated wires approach, our design is much simpler and achieves better performance. Further, our design solves several problems related to false negatives and correctness that are inherent in these earlier approaches. The second design is based on the use of repeated equalized transmission lines as a global interconnect. In contrast to earlier works that use transmission lines, our design is fully distributed and utilizes much more efficient repeated transmission line structures. Both of the proposed designs can be readily realized using conventional CMOS fabrication processes. This is a significant advantage over other approaches to the global communications problem, such as nanophotonics that require special fabrication steps or separate dies. In addition to these low-latency NoC designs, we have also developed a fast and accurate NoC-centric simulator, which accounts for interactions with applications, the processing cores, and the memory subsystem. For the purpose of evaluating NoC designs, our simulator achieves similar accuracies as conventional full-system simulators, but faster by orders of magnitude. 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 presents our NoC design based on asynchronous repeated wires and single-cycle multi-hop traversal.
- Chapter 4 presents our NoC design based on repeated equalized transmission lines.
- Chapter 5 presents our fast and accurate NoC-centric simulator.
- Chapter 6 concludes and discusses future research directions.
- Appendix A presents in greater details the asynchronous repeated wires that we use for the NoC design presented in Chapter 3.
- Appendix B presents in greater details the repeated equalized transmission lines that we use for the NoC design presented in Chapter 4.

Chapter 2

Background

In this chapter, we provide an overview of basic NoC concepts for the reader to better understand the thesis. We also provide necessary background about our target multi-processor system, evaluation methodology, and performance metrics that we use throughout this thesis.

2.1 Network-on-Chip Primer

Continuous improvements in CMOS technologies have already enabled us to have processors with hundreds of cores. These processors are typically referred to as chip-multiprocessors, multi-core processors, or simply multi-cores. We use these terms interchangeably throughout this thesis. For general-purpose multi-cores, any core can potentially communicate with any other core, and the traffic flows among them are often unstructured and unpredictable. Therefore, using dedicated or pre-configured connections among them is neither practical nor scalable. Alternatively, Network-on-Chip

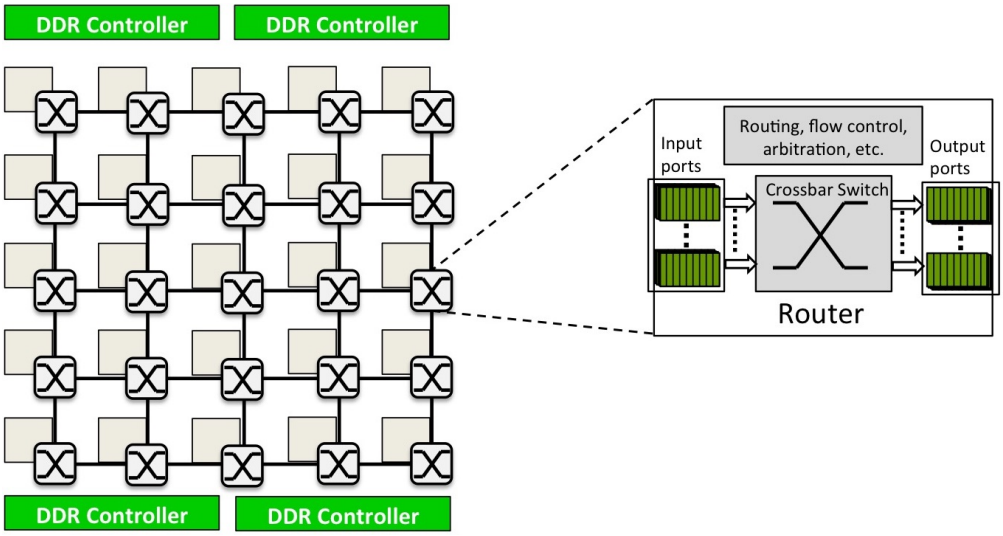


Figure 2.1: Mesh NoC overview.

(NoC) [25] has been proposed as a solution in which network resources (i.e., routers and links) are shared among the cores to multiplex different traffic flows[1] over them in different cycles. Fig. 2.1 depicts an example of an NoC that interconnects 25 cores for on-chip communication, with four memory controllers at the corners to manage off-chip data accesses. This NoC implements a two-dimensional mesh topology, which is widely used. In this figure, a router is attached to every core, and the routers are connected to the routers of adjacent cores via network links[2].

Using an NoC, communication starts with the source node encapsulating data into *packets*, which are then further broken into *flits* (or flow control digits). These flits in turn are injected into the network through the router attached to the source node. The flits are then routed through a series of intermediate routers, hop-by-hop, until they reach their the final destination. At each hop, the flits are buffered at the router inputs

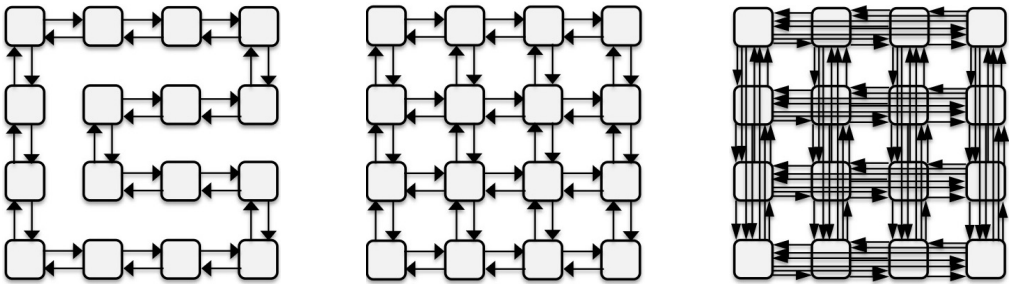
[1] A flow is a stream of interdependent bits from a source node to a destination node.
[2] We defer to Section 2.2 to discuss the core components.

and are then multiplexed over the outgoing network links to the next router that is enroute to the destination. There are several key design elements that characterize an NoC, including the network topology, the routing algorithm, the flow control mechanism, and the router microarchitecture. We discuss each of these design elements next.

2.1.1 Network Topology

The network topology determines the pattern that physical links connect routers to each other. Fig. 2.2 depicts three commonly used topologies, with different levels of complexity. Here, we discuss the trade-offs between the performance and cost for these topologies. For the topologies shown, the number of links is increasing from the left to the right. Fig. 2.2(a) depicts a ring in which each router is circularly connected to the adjacent routers located on its left and right. In a ring, the worst-case distance between two nodes is $N/2$ hops, where N is the number of nodes. Alternatively, a router can have connections to adjacent routers in four directions (i.e. North, South, West, and East) to form a mesh, as shown in Fig. 2.2(b). The network diameter of a mesh (worst-case distance between two nodes) is $2(\sqrt{N}-1)$, which is significantly smaller than a ring. As another example, a router can have connections to every other router in the same column and the same row along the X and Y dimensions to create a *flattened butterfly*, as shown in Fig. 2.2(c). In this case, the network diameter is reduced to just 2 at the expense of higher wiring cost and router complexity.

The choice of network topology is an important design choice that impacts other design choices (e.g., the routing algorithm). Each topology depicted in Fig. 2.2 can be a good choice, depending on the design objectives. For example, a ring is simple to



(a) Ring. (b) Mesh. (c) Flattened Butterfly.

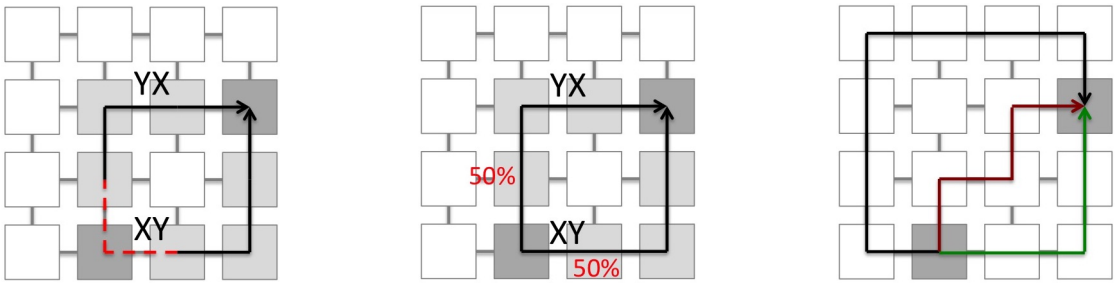
Figure 2.2: Example network topologies.

implement, but the average latency and effective throughput both degrade linearly with the network size. On the other hand, a flattened butterfly offers a much smaller network diameter and plenty of throughput, but has high implementation costs. In the middle, a mesh topology (as shown in Fig. 2.2(b)) offers a good balance between performance and cost; with respect to the network size, the average hop distance increases as a square root while the implementation cost grows linearly. Thus, the mesh topology is widely used in practice and is the most studied for NoCs.

2.1.2 Routing Algorithm

The choice of routing algorithm plays a vital role in the performance of an NoC. Depending on the network topology, multiple routing paths may exist between a source and a destination. To minimize latency, *minimal routing* is often preferred in which flits are routed along a shortest path from the source to the destination. Routing algorithms can also be categorized as *oblivious* or *adaptive*.

In the case of oblivious routing, the choice of routing paths is independent of the network state, which means the routing algorithm is unaware of congestions in the



(a) Dimension-ordered routing.

(b) O1TURN routing.

(c) Adaptive routing.

Figure 2.3: Routing algorithms for a mesh topology.

network. For mesh networks, the two most commonly used oblivious routing algorithms are *dimensioned-ordered routing* (DOR) [66] and *O1TURN* routing [62]. Both are minimal routing algorithms. There are two versions of DOR, XY-routing and YX-routing. In XY-routing, a flit always traverses first in the X direction (i.e., West or East), then makes a turn and traverses in the Y direction (i.e., North or South). In YX-routing, a flit instead first traverses in the Y direction, followed by the X direction. This is depicted in Fig. 2.3(a). Due to its simplicity, this routing algorithm is widely used. In O1TURN routing, both XY *and* YX routing are used with equal probability. This is depicted in Fig. 2.3(b). Surprisingly, this simple modification to use both XY and YX routing leads to much better performance than DOR in a number of adversarial traffic patterns. In fact, O1TURN can provably achieve near-optimal worst-case throughput [62].

Alternatively, in the case of adaptive routing, the choice of routing paths can dynamically change based on the network state to achieve better performance. The performance of an adaptive routing algorithm is largely determined by its ability to

accurately estimate congestion in the network, for which several approaches have been proposed [58, 30]. Fig. 2.3(c) illustrates some examples of adaptive routing paths.

2.1.3 Flow Control

Flow control determines the conditions when a flit at some router can be forwarded to the next router, and when it has to wait. Ideally, an efficient flow control mechanism should minimize wait times at low-loads and maximize throughput at high-loads. For a hop-by-hop NoC, flow control is applied to flits on a per-hop basis. A primary function of flow control is to check for the availability of free buffers at the next router. Lack of free buffers at the next router is a sign of contention. Therefore, the current hop should stop sending flits to this next hop to avoid a deadlock or livelock situation [27]. Further, we assume that flits cannot be dropped in the NoC. Thus, in the case of congestion, flow control should prevent upstream sources from injecting more flits into the network. By preventing an immediate upstream router from forwarding more flits, flow control creates a backpressure to throttle further upstream sources.

Packetization

We assume a packet-switched network for all hop-by-hop NoCs in this thesis. In packet switching, packets from different flows are time-multiplexed on the same physical links in the network. While packet sizes vary depending on the type of messages, which is usually defined by the application or cache coherence protocol (as we shall discuss in Section 2.2), the flit size is a function of the link width, which is a design-time parameter. Typically, the flit width is the same as the link width (e.g., 128 bits). For traversals over

network links, a packet is divided into flits. Depending on its relative position within a packet, a flit is either a *head*, *body*, or *tail* flit. The head flit carries important information about the packet, including address information and routing type. In the case of single-flit packets, the flit is both its head and tail. Flits are routed hop-by-hop until they reach their destination.

Flow Control Mechanisms

For packet-switched networks, there are several well-known flow control mechanisms for allocating buffers and links. They differ by their level of granularity.

- *Store-and forward*: Each router waits until an entire packet has arrived before it considers forwarding the packet to the next router. The packet can only be forwarded to the next router if the next router has the buffer space to accept the entire packet. Thus, storage is allocated at a packet granularity.

- *Virtual cut-through*: This flow control mechanism allows flits to be forwarded to the next router before the entire packet has arrived, but storage is still allocated at a packet granularity. That is, the next router still has to have the buffer space to accept an entire packet. This just has to be checked for the head flit.

- *Wormhole*: This flow control mechanism also allows flits to be forwarded to the next router without waiting for an entire packet to arrive. Wormhole flow control differs from virtual cut-through flow control in that routers can have buffer storage smaller than the size of packets, which means a flit can only be forwarded to the next router if the next router has the buffer space to receive it. This must

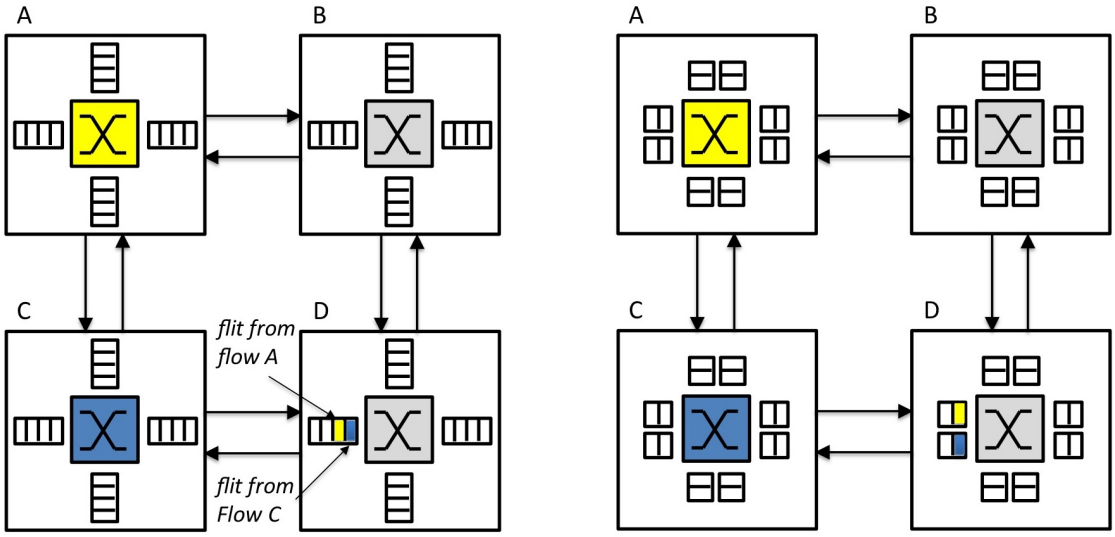
be checked for every flit. In wormhole flow control, storage is allocated at a flit granularity.

In this thesis, we assume *virtual cut-through* flow control.

Virtual Channel

Virtual channels are used to solve a problem called *head-of-line blocking*. Consider the example shown in Fig. 2.4(a). In this example, flits from different flows are all queued one behind another in the same FIFO. Suppose routers A and C both have a flit buffered at the input of router D, with the flit from router C at the *head-of-line*. Suppose flow C is blocked at router D due to insufficient buffer space at its next router, but suppose the next router for flow A has plenty of buffer space for flow A to proceed. In this case, the flit from flow A is nonetheless blocked from proceeding because it stands behind the head-of-line flit from flow C, which is blocked from proceeding. This head-of-line blocking can potentially lower the network throughput.

Virtual channels alleviate the head-of-line blocking problem by associating separate *logical* queues for different flows at a router. Fig. 2.4(b) depicts the same example, but using two virtual channels at each input port. As we shall see in Section 2.1.4, a router can pick either virtual channel to serve. As such, the flit from flow A is no longer blocked by the flit from flow C and therefore can be served and leave router D. In general, a VC gets allocated on the head flit, and the VC gets released when the corresponding tail flit leaves. Flits within the same packet will traverse the same path because they use the same VCs at the intermediate routers, which guarantees that flits will be kept in order.



(a) Head-of-line blocking without VCs.

(b) Virtual channels.

Figure 2.4: Virtual channel flow control (flow C is blocked at router D).

Buffer Management

In this thesis, we assume virtual cut-through flow control. In this method, a head flit can only depart a router if the next hop has a free VC to allocate to it. We assume that each VC buffer is large enough to store an entire packet if necessary. Therefore, the body and tail flits of a packet do not need to check for credit availability at the next hop.

Virtual Networks (VNets)

Cache coherence protocols typically use different *message classes* for different types of messages to avoid deadlocks. For example, a protocol can require messages from different message classes to use different sets of queues within the network. This can be implemented using *virtual networks* (VNets) by allocating each VNet a separate

pool of buffers for its VCs, but the VNets are still multiplexed over the same physical network. Besides the avoidance of deadlocks, VNets can also be useful for implementing different quality-of-service for different message classes.

2.1.4 Router Microarchitecture

Fig. 2.5 depicts the microarchitecture of a 5-port NoC router for a mesh, which forms the baseline for the NoC designs presented in this thesis. An NoC router is responsible for buffering incoming flits and multiplexing them to the output ports at every cycle. Each input port has buffers that are divided into separate VCs, with each VC implementing a FIFO queue. In the case of multiple VNets, the input port buffers are further divided among VNets and VCs within VNets (this is not shown in Fig. 2.5). To provide connectivity from any input port to any output port, input ports are connected to a crossbar switch on one side, and output ports on the other side. In any given cycle, each input port can only be matched to one output, and each output port can only be matched to one input.

As discussed earlier, in an NoC, each packet is divided into flits. Based on its location, a flit is either a head, body, or tail flit. The head flit is the first flit in a packet and carries destination information. It is followed by one or more body flits and a tail flit[3]. For a virtual channel router, flits go through the following steps at every hop. Among these steps, body and tail flits do not need to go through route compute or VC allocation since they follow the path of their head flit.

- *Buffer Write (BW)*: Incoming flits are written into their VC buffer.

[3]In the case of single-flit packets, the flit is both its head and tail.

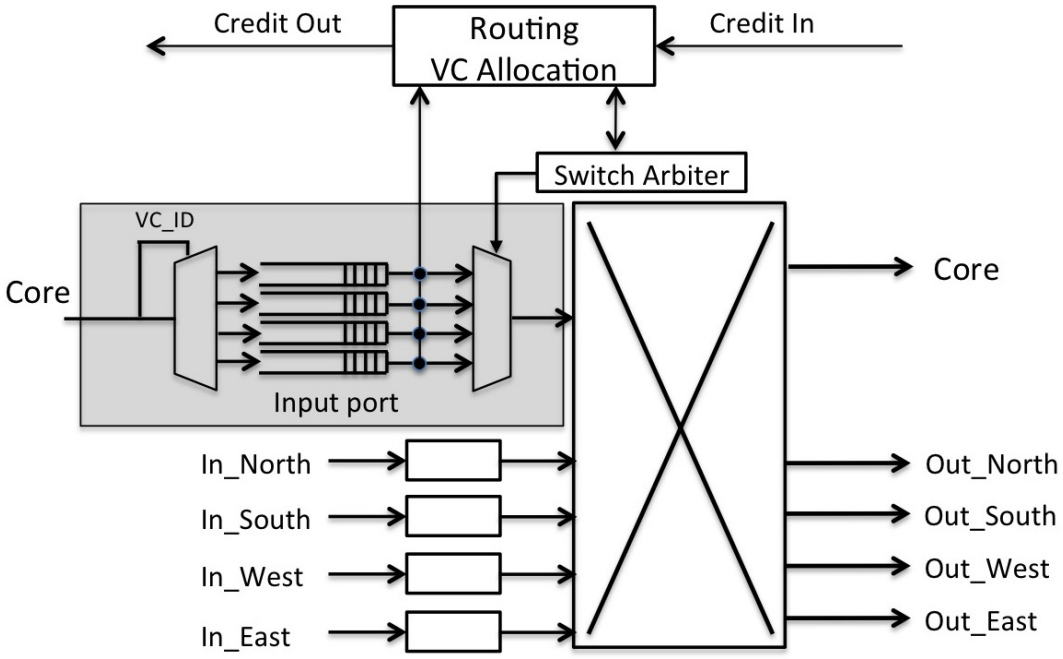
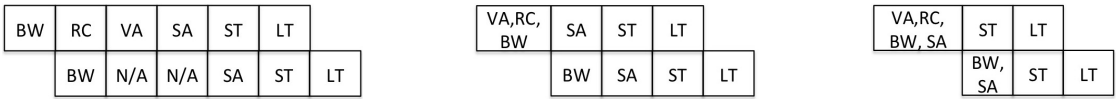


Figure 2.5: Router microarchitecture for a 2D-mesh.

- *Route Compute (RC)*: RC selects the departing output port based on the destination information and the routing algorithm. For example, if the destination is (dx, dy) and the current hop is (x, y) such that $dx > x$, then the departing output port would be the *East* port in the case of XY-routing.

- *VC Allocation (VA)*: All flits need a guaranteed VC at the next-hop router before proceeding. Therefore, a VC has to be allocated before a head flit arrives. Body and tail flits simply use the same VC as their head flit. In this thesis, we assume virtual cut-through flow control in which the allocated VC provides enough space to buffer an entire packet.

- *Switch Allocation (SA)*: At each cycle, VCs must arbitrate access to the crossbar switch. For an n-port router with v VCs per input port, SA is essentially a matching



(a) 5 Stages. (b) 3 Stages. (c) 2 Stages.

Figure 2.6: Router pipeline designs. The number of stages shown are for the number of router stages, which does not include the link traversal (LT) stage.

problem from nv contenders at the inputs to n outputs. This is often achieved in two steps using a separable allocator approach [57] in which a VC is first selected at each input port using a $v:1$ arbiter (the selection of VC also selects an output port for the input). Then, among the competing input ports to an output port, an $n:1$ arbiter is used in the second step to select the final input/output crossbar matchings.

- *Switch Traversal (ST)*: Winners of SA traverse the crossbar in this step by forwarding the head-of-queue flits from the corresponding winning VCs.

- *Link Traversal (LT)*: Flits coming out from ST then traverse the link to the next router.

In a traditional NoC router design, the above steps are mapped to a 5-stage pipeline, as shown in Fig. 2.6(a). In this pipeline, it takes a flit five cycles to make one hop in the NoC. There are many proposals to reduce the number of pipeline stages by allowing parallel or faster execution of the above steps. One approach is to merge stages together by using faster components [47, 56]. Another approach is to bypass some stages by fast-forwarding routing information to downstream routers [49, 48]. Yet another

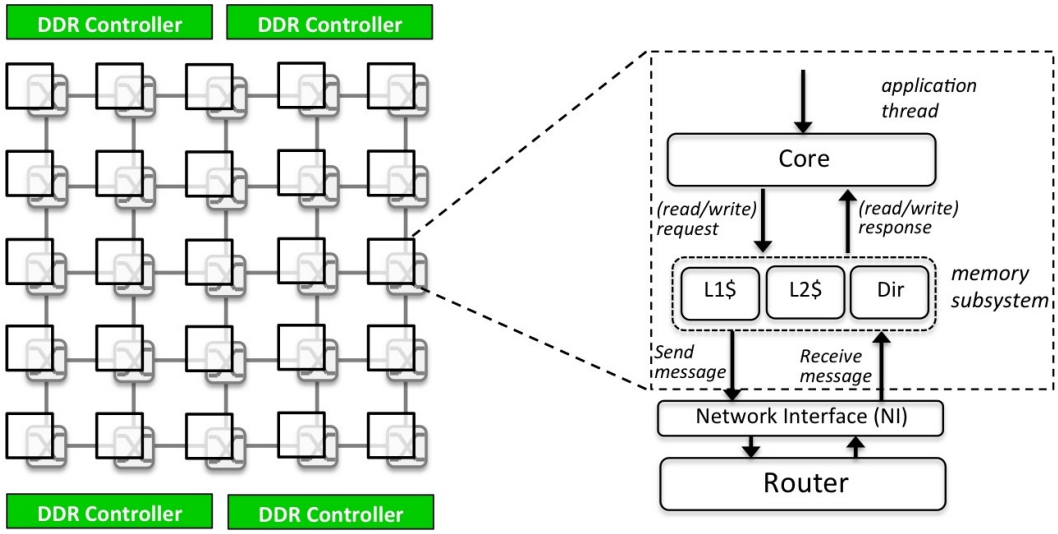


Figure 2.7: Shared memory chip-multiprocessor architecture.

direction is to speculatively execute stages in parallel, and roll back and re-execute the stages sequentially when speculation fails [52, 54]. All these approaches help significantly to reduce the number of pipeline stages. For example, a 1-cycle router has been shown [56]. However, in this case, a flit still needs to spend 2 cycles at each hop even in the absence of contention (1-cycle router + 1-cycle LT). Fig. 2.6 depicts two other router pipeline designs that have been proposed in the literature.

2.2 System Architecture

We assume a tile-based shared memory multiprocessor architecture for all designs in this thesis. Fig. 2.7 illustrates the architecture in which each tile comprises a core, a local L1 cache, a slice of the L2 cache, and a directory to track sharers of cache-lines [64]. We assume a cache coherence protocol is used to keep information in the private L1 caches consistent and coherent with the shared data in L2.

2.2.1 Message Flows through the Network

In this section, we briefly describe how messages are generated and how they flow through the network. The message flow starts by the execution of a memory instruction at a processor core for an address in the memory subsystem. For example, in the case of a read request, the core first checks its local L1 cache for the data. If there is a cache miss, the cache coherence protocol has to resolve the cache miss by generating a series of cache coherence messages that flow through the network before it can return control to the core. For example, the cache coherence protocol has to first consult the directory that is home to the read address to find the remote location of a valid copy of the data, if it exists somewhere on-chip. Then, it uses that information to retrieve the cache line and updates the L1's local copy. This procedure may generate multiple messages that flow through the network in response to the read request. Similarly, when the L1 cache receives a write request, the cache coherence protocol has to first use the directory that is home to the write address to invalidate the other local copies of the cache line. This again may generate multiple messages that flow through the network in response to the write request. In either case, the memory subsystem design dictates the pattern and volume of messages that flow through the network[4]. Thus, the traffic pattern observed on the NoC is largely characterized by the cache coherence messages that flow through the network, not just by the application's memory access pattern.

[4]Addressing different memory subsystem designs is not a focus of this thesis. We refer the interested reader to [79, 64] for a more thorough discussion.

2.2.2 Memory Subsystem

Cache coherence protocol

We use a full-state cache coherence protocol for evaluations of the designs in this thesis. This class of protocols includes an extra *owned* state[5], which helps to avoid the write-back of dirty cache lines to the main memory. As a result, running applications are less likely to idle waiting for off-chip memory updates.

We consider both shared and private L2 designs in our evaluations. For a shared model, the address space is evenly distributed across all tiles. For a private model, each L2 portion is the home for the full address space. The directories are also distributed; they keep the state information of the associated L2 piece located at the same tile.

Network interface

In a shared memory multiprocessor, network traffic corresponds to memory subsystem messages (i.e., L1/L2/directory requests or responses). The network interface (NI) is responsible for sending and receiving these messages at each node. On the sending side, the NI first packetizes messages into packets, which are then further broken into flits before they are injected at the source node into the network. On the receiving end, the NI reassembles the flits received at the destination into packets, which are then further reassembled into messages. Although flits are always delivered in order (as discussed in Section 2.1), packets can arrive out-of-order at the destination because of the use of different paths or virtual channels. Therefore, the NI may need to reorder them before a message can be reassembled and delivered to the receiving entity (e.g., a cache

[5]The *owned* state represents the situation in which a cache line is both modified and shared. The other coherency states are *modified*, *exclusive*, *shared*, and *invalid*.

controller).

Unless otherwise specified, flits are 128-bit wide and cache lines are 64B in size. We classify memory subsystem messages as either control or data messages, with the corresponding lengths of 1 or 5 flits[6]. For the NoC designs in this thesis, control messages use a VNet with 1-flit deep VC buffers, while data messages use a VNet with 5-flit deep VC buffers.

2.3 System Evaluation

Computer architects need accurate network simulations to develop effective NoC solutions and evaluate design trade-offs. In this thesis, we use both full-system simulations of real applications as well as synthetic network simulations of synthetic traffic patterns to evaluate designs. In this section, we present the system configurations that we use for our evaluations as well as the evaluation methodology and performance metrics that we use to evaluate our designs. We also describe the tools that we use for our evaluations.

2.3.1 System Configurations

Real applications: We use applications from the PARSEC [14] and SPLASH-2 [75] benchmark suites for real traffic evaluations. Unless otherwise specified, the results represent the parallel sections of the benchmark applications. In some cases in Chapter 5, we present results for finer granularities to show the behavior of an application

[6]For the cache coherence protocol used in this thesis, control messages correspond to *request_read*, *request_write*, *invalidation*, *acknowledge*, and *unblock* messages. Data messages correspond to *response* types.

in particular sub-regions of interest. The number of parallel threads is always set to be equal to the number of cores (i.e., 16 threads for 16 cores, 64 threads for 64 cores).

Synthetic traffic: We use the following three synthetic traffic models in our evaluations to stress the designs under different injection rates:

- *Uniform Random*: Under this model, a source will generate traffic to a destination chosen at uniform random. This model represents traffic with destinations located at moderate distances.

- *Tornado*: Under this model, a source at location (x, y) will generate traffic to a destination at location (dx, y), where $dx = (x + 3) \mod k$, where k is the radix of the mesh network (e.g., $k = 8$ for an 8×8 mesh). This model represents traffic in which each node only communicates with nearby neighbors, but the traffic pattern is nonetheless difficult to handle.

- *Bit Complement*: Under this model, a source at location (x, y) will generate traffic to a destination at location (dx, dy), where dx and dy are bitwise complements of the binary encodings for x and y, respectively (e.g., if $x = 000$, then $dx = 111$). This model represents traffic with destinations that are on the opposite side of the chip. This traffic pattern has the highest average distance between sources and destinations among the three synthetic traffic models.

Memory and core configurations: For cache coherence, we employ a MOESI directory protocol [32], which is a full-state protocol. For the L2 cache, we use both shared and private cache models in our experiments. The size of the L2 cache is between 2-16MB, which is assumed to be evenly distributed across the tiles[7]. Unless otherwise

[7]For a 16-core system and a 2MB L2, each tile has a 128KB portion of the cache capacity.

specified, we assume 32KB I&D L1 caches per tile. The number of directories is equal to the number of tiles – each directory is co-located with the corresponding L2 cache slice. We assume a single virtual address space for all on-chip caches. A TLB cache performs translations to/from off-chip physical addresses. Unless otherwise specified, we assume we have two DRAM controllers that are embedded in the tiles located at the opposite corners of the chip. The choice of DRAM controller to use is assumed to be chosen at random. Unless otherwise specified, we assume a core model similar to an in-order SPARC core.

NoC configurations: For the NoC designs, we consider three VNets to carry control and data messages separately (i.e., L2 requests, directory requests, and directory responses). As mentioned above, the VC buffers for control and data messages have depths of 1 or 5 flits, depending on the VNet, to accommodate an entire packet (virtual cut-through). Depending on the experiment, the number of VCs is chosen accordingly. For mesh NoCs, we use symmetric meshes when possible (e.g., 4×4 for 16 nodes, 8×8 for 64 nodes). In all experiments, flits are 128-bit wide, unless otherwise specified.

2.3.2 Evaluation Methodology

Performance evaluation: We use gem5 [15] for full-system simulations and Garnet [7] for NoC performance evaluations. We also use synthetic network simulations with synthetic traffic models, as described above, to evaluate designs. In particular, we use a cache coherency-aware synthetic network simulator called Synfull [11] for our evaluations under synthetic traffic patterns. The synthetic network simulation approach allows us to go beyond the 64-core limit that gem5 can handle. The simulation runs

are configured to match the system specified in Section 2.3.1. We mostly use network latency as the main evaluation metric since it is the primary performance objective that we are trying to improve in this thesis. To evaluate network throughput, we primarily use synthetic network simulations, which allow us to freely increase the injection rates to find the saturation points of the designs. We define the *saturation point* of the network as the injection rate at which the average network latency exceeds 3× the low-load latency. We assume all network nodes have the same injection rate in our synthetic network simulations.

Energy and area evaluations: All energy and area evaluations in this thesis are based on a 45nm technology. Throughout Chapters 3 and 4, we use DSENT [67] to calculate energy and area results. We also use the Synopsys Design Compiler to synthesize implementations for energy and area results as well. By default, all presented power results are for dynamic energy. We intentionally exclude leakage power from our energy results because of its high contribution to the total energy at low injection rates. This way, we can have a clearer picture of the improvements made by our proposed designs. Having said this, the leakage problem can be improved significantly by the use of techniques like power gating, which by itself is a different research topic and beyond the scope of this dissertation.

2.4 Chapter Summary

In this chapter, we provided necessary background on NoC for the reader to better understand the various techniques presented in this thesis. The chapter also presented necessary background about our target multi-processor system, evaluation methodology,

and performance metrics that we use to evaluate our work in this thesis.

In the next chapter, we present an NoC design that uses asynchronous repeated wires to achieve single-cycle multi-hop traversals. This design aims to realize the low-latencies of dedicated wire connections while retaining the throughput, flexibility, and scalability of a shared NoC.

Chapter 3

Single-Cycle Multi-Hop NoC Designs

3.1 Introduction

This chapter explores an NoC architecture that allows flits to dynamically create and traverse *multi-hop* routes within a single-cycle, potentially all the way from the source to the destination. It builds on the work by Krishna et al. [45], who proposed an innovative NoC architecture called SMART (Single-cycle Multi-hop Asynchronous Repeated Traversal) based on the use of *single-cycle multi-hop repeated wires*. They showed that an electrical signal can traverse multiple hops within a single clock cycle if the wires are repeated with appropriately sized drivers and properly spaced. The use of these repeated wires enables a flit to bypass entirely all the pipeline stages and queuing at the intermediate routers. The reader can refer to Appendix A for a detailed presentation about the implementation of these single-cycle multi-hop repeated wires. The focus of

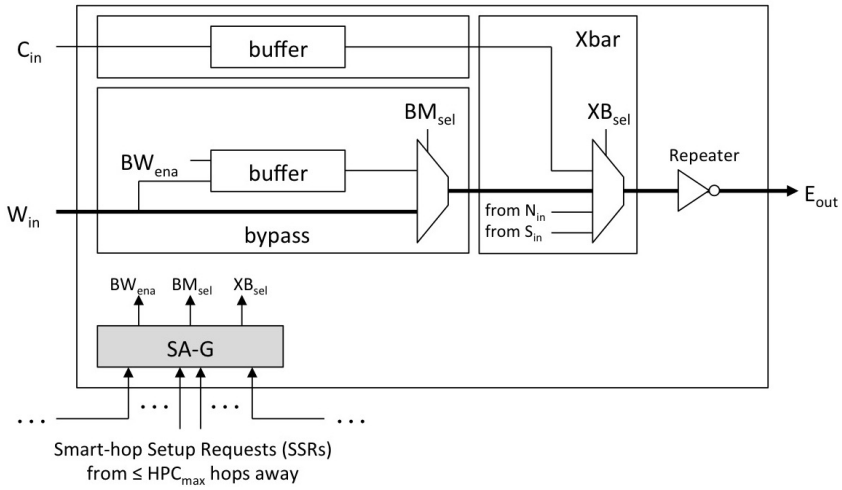


Figure 3.1: SMART router microarchitecture.

this chapter is on a new SMART-based NoC design called SHARP, which stands for *Smart-Hop Arbitration Request Propagation*. It is based on a *propagation-based SSR arbitration* mechanism that avoids several inherent shortcomings of SMART, including quadratic complexity and throughput loss due to false negatives.

3.2 SMART NoC

This section describes the key elements of SMART and how it operates. Fig. 3.1 depicts the microarchitecture of a 5-port SMART router for a mesh network. For simplicity, only the $Core_{in}$ (C_{in}), $West_{in}$ (W_{in}), and $East_{out}$ (E_{out}) ports are shown in details[1]. All other input ports are identical to W_{in}, and all other output ports are identical to E_{out}.

In a SMART NoC, conventional clocked link drivers are replaced with asyn-

[1]SMART uses a mesh topology for connecting routers; hence, each router has five input/output ports (*West*, *East*, *North*, *South*, and *Core*).

chronous repeaters at every hop, allowing a flit to traverse multiple hops in a single clock cycle. To enable a single-cycle multi-hop path, SMART adds an alternative data-path to each router to allow a flit to bypass the entire router pipeline and go directly to the next router. This is depicted in Fig. 3.1 with a bold line going from W_{in} to E_{out}.

To facilitate the configuration of a bypass path, each router has a 2:1 multiplexer (mux) at the input of the crossbar to choose between the local buffered flit and the bypassing flit. On a bypass selection, the input buffer will be disabled to avoid latching the bypassing flit, and the crossbar will be configured to connect the input to the proper output port. To control the input buffer, the bypass 2:1 mux, and the crossbar, each router provides a Buffer Write enable (BW_{ena}) signal, a Bypass Mux select (BM_{sel}) signal, and a Crossbar select (XB_{sel}) signal, respectively. The crossbar output is connected to the next router via a repeater that is sized to drive the link as well as the two muxes (2:1 bypass and 4:1 crossbar) at the next router.

An example of a multi-hop traversal (called a SMART-hop) is illustrated in Fig. 3.2: a flit from router R20 travels three hops within a single-cycle, until it is latched at R23. The crossbar at R20 is set to connect C_{in} to E_{out}. The crossbars at R21 and R22 are set to connect W_{in} to E_{out}, with their BM_{sel} set to choose bypass over local. At R23, its BW_{ena} is set to latch the incoming flit. A SMART-hop path can thus be created by appropriately setting BW_{ena}, BM_{sel}, and XB_{sel} at intermediate routers.

To setup SMART-hops, SMART performs switch allocation in two stages: local switch allocation (SA-L) and global switch allocation (SA-G). The SA-L stage is identical to the switch allocation step in a conventional NoC router in which buffered flits at a router arbitrate among themselves to gain access to the output ports. For each winning

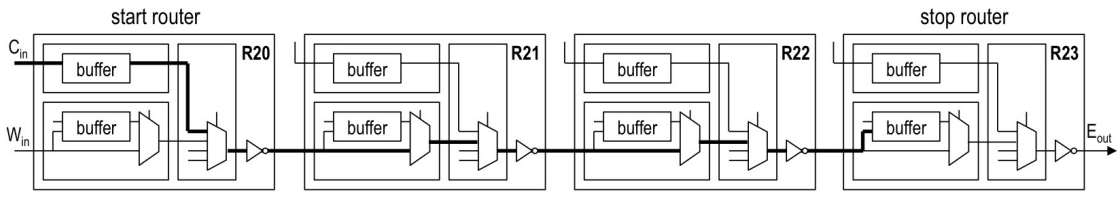


Figure 3.2: Traversal over a SMART-hop path.

buffered flit, the source router broadcasts a corresponding Smart-hop Setup Request (SSR). Each of these SSR is sent through dedicated multi-drop wires that are repeated from the source router to all intermediate routers up to HPC_{max} hops away along a chosen path to the destination, where HPC_{max} is the maximum number of hops that a flit can traverse in a single cycle[2]. The chosen path can either be an XY-path or YX-path.

Upon receiving the SSRs, the recipient routers conduct a second switch allocation stage called SA-G. In this stage, the recipient router arbitrates among the received SSRs to setup the different parts of a router to operate in bypass (by setting BM_{sel} to bypass) or stop mode (by setting BW_{ena} to latch the incoming flit). This SA-G stage comprises the following two parts:

- *Input arbitration*: Routers first arbitrate among multiple SSRs to choose the highest priority requests (per input port).

- *Output arbitration*: A four-way arbitration is performed after input arbitration to resolve possible conflicts at each output port (e.g., at output E_{out}, winning SSRs from W_{in}, N_{in}, and S_{in}, and possibly a local request from C_{in} are considered to

[2]In [45], it was shown that an $HPC_{max} = 8$ is best achieveable HPC_{max} for a 2D configuration when energy is taken into consideration. Besides the delays through the repeated wires, the best achieveable HPC_{max} also has to take into account the delays through muxes (bypass and crossbar muxes) at intermediate routers as well as the complexity of the SA-G arbitration logic.

Figure 3.3: SMART pipeline. * indicates head-flit only. SHARP follows the same pipeline, but performs propagation-based SSR arbitration in SA-G instead of parallel-based SSR arbitration.

select a winner).

When multiple SSRs from multiple sources arrive at the same time, the recipient router can use two priority policies for allocation: (a) *prio_local*, which gives nearby requests higher priority, and (b) *prio_bypass*, which gives remote requests higher priority. When arbitrating among SSRs from routers that are the same distance away, they can be prioritized based on direction. In [45], it was proposed to choose straight-hops > left-hops > right-hops, where straight, left, and right are relative to the input/output port.

After the switch allocation stages, a flit that won both SA-L and SA-G at its source router will proceed and bypass all the intermediate routers in which its SSR won the corresponding SA-G step. Therefore, a flit needs to spend at least two cycles at the source router before traversing a SMART-hop. The SMART router pipeline is shown in Fig. 3.3. In a nutshell, a SMART-hop traversal is performed as follows:

- At cycle $t-1$, source routers arbitrate the buffered flits and choose a local candidate for every input/output ports (SA-L).
- At cycle t, SA-L winners broadcast SSRs to routers along their desired path using dedicated wires, and consequently, recipient routers perform SA-G among them by

employing a fixed priority scheme and configurating routers accordingly.

- At cycle $t + 1$, SA-G winners traverse the bypass paths that have been setup in a single cycle.

As we shall see in Section 3.4, our NoC design called SHARP follows the same pipeline, but it uses a propagation-based SSR arbitration scheme instead of the parallel-based SSR arbitration scheme used by SMART, which addresses a number of shortcomings of SMART, as described next.

3.3 Shortcomings of SMART

In this section, we describe three inherent shortcomings from which SMART suffers. The first shortcoming is due to the fact that SMART employs a *parallel* SSR arbitration mechanism in which all SSRs that are within HPC_{max} hops away must be considered, which means each router must arbitrate among up to $HPC_{max}(2HPC_{max} - 1)$ SSRs at each input port. This quadratic complexity is expensive both in terms of area and power. The second shortcoming is due to the fact that *all* routers independently make their own allocation decisions in *parallel*, without knowledge of allocation decisions made by upstream routers. This parallel SSR arbitration mechanism leads to a problem called *false negatives* that causes throughput loss. Finally, as noted in [45], all routers need to enforce the same priority, either *prio_local* or *prio_bypass*, to ensure correctness. Incorrect behavior can result if a mixed priority scheme is used. This is again a consequence of routers making independent allocation decisions in parallel. These three shortcomings are discussed further below.

3.3.1 Quadratic complexity of parallel SSR arbitration

For the SMART-2D[3] design [45], the SA-G stage considers up to $HPC_{max}(2\ HPC_{max}-1)$ SSRs per input port from routers up to HPC_{max} hops away. Fig. 3.4(a) illustrates this for the W_{in} port of router R_{23} with $HPC_{max} = 3$. Note that although the figure shows each source router is connected to R_{23} via a single SSR, sources on the same row may contribute multiple SSRs[4] at W_{in} (i.e., R_{23} may receive up to 3 SSRs from R_{21} and up to 5 SSRs from R_{22}). This is because every path of length $HPC_{max} = 3$ that has its turn position at or after (2, 3) will also go through R_{23} (one dedicated SSR for each). Together, R_{23} may receive up to 15 SSRs at W_{in} from sources within $HPC_{max} = 3$ hops away. A similar analysis can be done for all other directions of R_{23}, and generally for all nodes in a mesh topology.

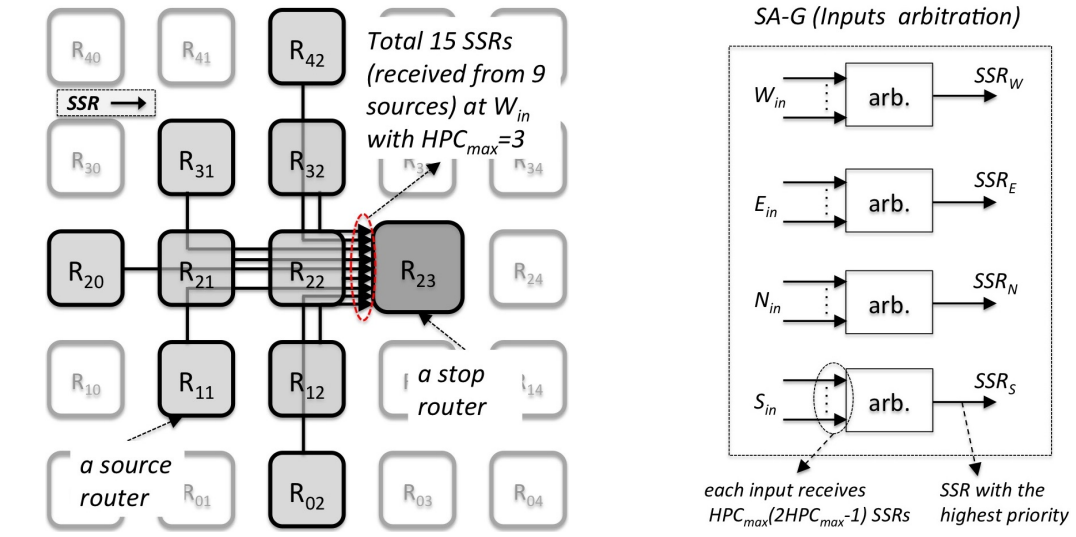
Upon receiving the SSRs, the SA-G stage arbitrates among them to choose the highest priority request at each input port. This is depicted in Fig. 3.4(b) for each input in the four mesh directions (*West*, *East*, *North*, *South*). All these dedicated SSR links together impose an $O(HPC_{max}^2)$ wiring cost on the control plane.

Besides a quadratically increasing wiring cost, the area and energy consumption for the SA-G stage also grow at $O(HPC_{max}^2)$ as well. To understand this, the required logic in SMART for input arbitration, output arbitration, and local configurations are considered, as shown in Fig. 3.5[5]. The input and output signals correspond to the ones shown in the router in Fig. 3.1. Fig. 3.5 is similar to the one shown in [45], which focuses

[3]In [45], a SMART-1D design was proposed as well in which SMART-hops cannot make turns. SMART-2D allows one turn in a dedicated path. Unless otherwise noted, SMART refers to SMART-2D in this dissertation.

[4]At most one SSR originating from the same source can be active per cycle.

[5]The implementation shown is for *prio_local*. The implementation for *prio_bypass* is similar, but not discussed.



(a) (b)

Figure 3.4: Broadcast SSRs in SMART-2D. (a) SSRs received at an input port. (b) Input arbitration in the SA-G stage.

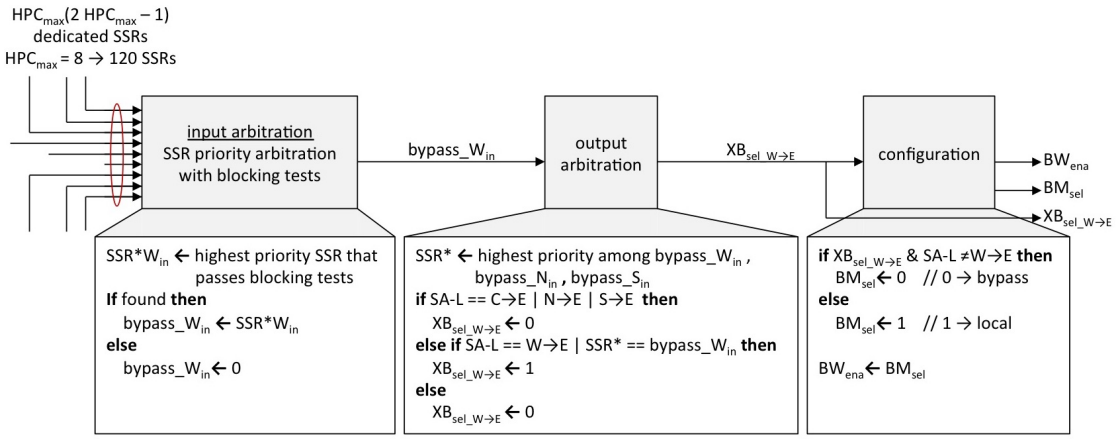


Figure 3.5: Implementation of parallel-based SA-G for *prio_local* in SMART, focusing on W_{in} and E_{out} [45].

on the logic for SSRs coming in at the input port W_{in} for bypass to the output port E_{out}. We modified the figure from [45] to emphasize two additional details: (1) to determine if an SSR has to terminate or prematurely stop at the current input port, additional *blocking tests* besides checking for a *free VC* at the next-hop are required; (2) besides checking for local requests, the output arbitration step also has to check for competing SSRs from the input ports N_{in} and S_{in}. The functions in Fig. 3.5 are explained as follows.

Input arbitration

The arbitration process starts with an examination of up to $HPC_{max}(2HPC_{max}-1)$ SSRs at each input port to select the highest priority request. Depending on the priority scheme (i.e., *prio_local* or *prio_bypass*) a single SSR winner is chosen by comparing *distances* and *directions* of the SSRs relative to the recipient router. In particular, in the case of the *prio_local* policy, the *closest* SSR to the recipient router wins the arbitration; in the case of the *prio_bypass* policy, the *farthest* SSR wins the arbitration. In the case of equal-distance SSRs, the authors in [45] proposed to break ties based on directions: straight $>$ left-turn $>$ right-turn. We adopt the same tie-breaker policy in our work.

To determine if a given SSR is an *active* request, the recipient router examines the length information (in hops) that is encoded in the SSR. This length information, referred to as the *hop_num*, indicates the number of hops that the corresponding flit wishes to travel. The recipient router uses this length information, which can be encoded in $\log_2(1 + HPC_{max})$ bits, to determine if the SSR terminates earlier.

Besides prioritizing the SSRs based on distances and directions, input arbitration

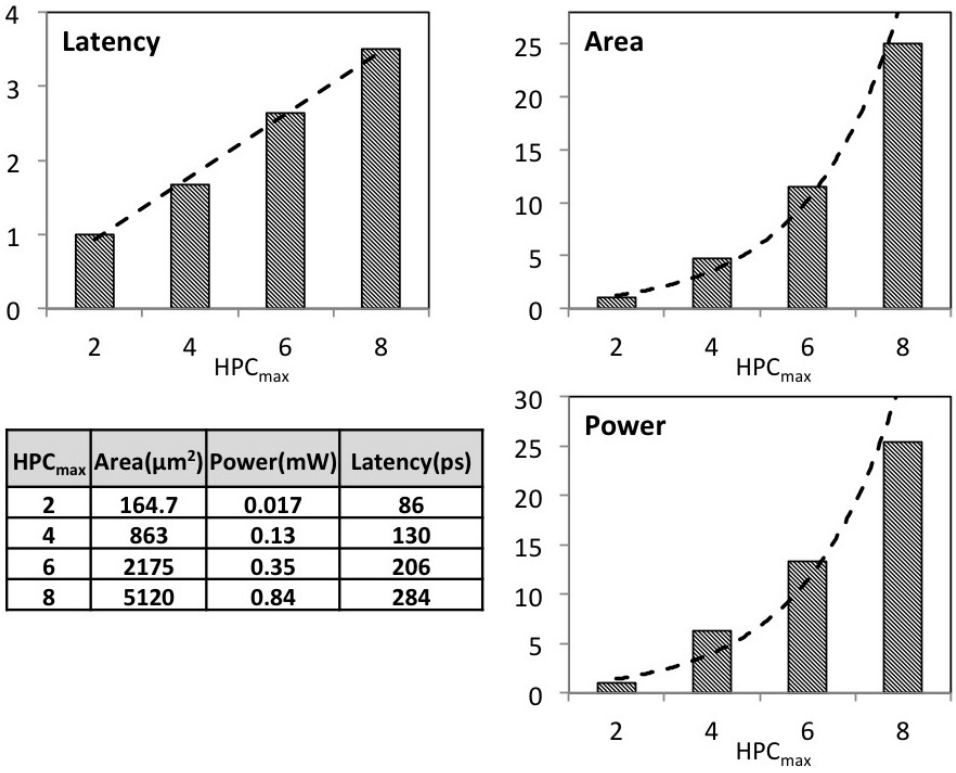
in SMART also performs *blocking tests* to see if an SSR should be considered for bypass to an output. There are several reasons why an SSR at an input port does not need to be considered for bypass, each of which must be checked during input arbitration.

- If the *prio_local* policy is used, then the arbiter checks to see if there is a local request that won the SA-L stage for the input port, since this local request would have higher priority than requests coming from one or more hops away.
- The arbiter also checks to see if there is a *prematurely* stopped flit from the same source as a given SSR among its buffered flits. If there is, then the SMART-hop path must stop at this input port to ensure the ordering of flits at the destination.
- In addition, the arbiter checks for the availability of a free VC at the next-hop router for a given SSR.
- Finally, the arbiter checks to see if the current router is the final destination for a given SSR.

Together, input arbitration selects the highest priority SSR that passes the above blocking tests for further arbitration at the corresponding output port.

Output arbitration and configuration

Once the winners of the input arbitration part are determined, output arbitration is performed to resolve possible conflicts at each output port (using the same priority policy). Then, the control signals (i.e., BW_{ena}, BM_{sel}, and XB_{sel}) are appropriately set to enable flit traversals in the next cycle.



HPC_{max}	Area(μm^2)	Power(mW)	Latency(ps)
2	164.7	0.017	86
4	863	0.13	130
6	2175	0.35	206
8	5120	0.84	284

Figure 3.6: SSR arbitration costs.

Complexity

The above analysis for the SA-G logic provides some insights. First, the selection of the highest priority request at each input port is related to the number of SSRs received, which in turn is quadratic with respect to HPC_{max}. However, the complexity of the other arbitration parts is constant, independent of HPC_{max}.

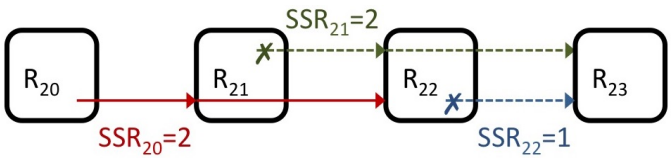
To evaluate the arbitration cost in practice[6], we use the Synopsys Design Compiler to synthesize the SA-G logic in a TSMC 45nm technology. Fig. 3.6 depicts the area, power, and latency costs for four HPC_{max} values. In this figure, the table shows the absolute values, and the bar graphs are based on normalized values (normalized to

[6]We defer the reader to [22] for the wiring cost analysis.

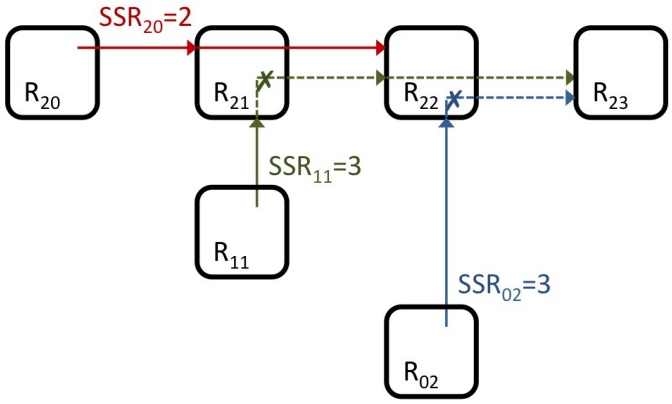
$HPC_{max} = 2$). Indeed, the results clearly show quadratic increases in area and power, and near linear increases in latency, with respect to increasing HPC_{max}. For a large HPC_{max} value, the arbitration logic consumes a significant amount of the area and power budgets of a router and imposes a high latency overhead on the control-path.

3.3.2 False negatives and throughput loss

As noted in [45], the parallel SSR arbitration scheme used by SMART leads to *false negatives*, a situation in which a router has been setup to bypass or stop an expecting flit, but no flit arrives. This can happen because all routers are *independently* making their own allocation decisions at the same time, *without* knowing the allocation decisions that other upstream routers will make. We first illustrate a false negative scenario under the *prio_bypass* policy. Fig. 3.7(a) depicts an example in which router R_{20} sends an SSR to R_{22}, and both routers R_{21} and R_{22} send SSRs to R_{23}. At R_{21}, SSR_{20} wins output arbitration over SSR_{21}, so the corresponding flit will be able go all the way to its stop router R_{22}, as shown in the solid red line from R_{20} to R_{22}. However, even though SSR_{21} lost output arbitration at its start router, it wins output arbitration over SSR_{22} at R_{22}, which prevents R_{22} from sending its own flit. That is, R_{22} could have sent a flit to its stop router R_{23}, as shown in the dashed blue line, but cannot because R_{22} has no way of knowing that SSR_{21} has lost output arbitration at R_{21}. Therefore, even though R_{23} is expecting to receive a flit from R_{21}, the flit does not arrive (i.e., a false negative occurs). 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.



(a) Under *prio_bypass*.



(b) Under *prio_local*.

Figure 3.7: Throughput loss due to false negatives.

We next illustrate a false negative scenario under the *prio_local* policy. Fig. 3.7(b) depicts an example in which router R_{20} sends an SSR to R_{22}, and both routers R_{11} and R_{02} send SSRs to R_{23}. For the *prio_local* policy, SSR_{20} wins output arbitration over SSR_{11} at R_{21}, even though they are equal-distance to R_{21}. This is because straight-hops have higher priority than right-turn-hops in the case of an equal-distance tie breaker [45]. This enables R_{20} to send a flit all the way to its stop router R_{22}, as shown in the solid red line in Fig. 3.7(b). At R_{22}, SSR_{02} loses output arbitration to SSR_{11} because the straight arrival of SSR_{11} into R_{22} has higher priority than the right-turn arrival of SSR_{02}. This forces the flit from R_{02} to prematurely stop at R_{22}, even though the output link from R_{22} to R_{23} is idle. That is, R_{02} could have sent a flit all the way to its stop router R_{23}, as shown in the partially dashed blue line, but cannot because R_{22} has no way of knowing that SSR_{11} has lost output arbitration at R_{21}.

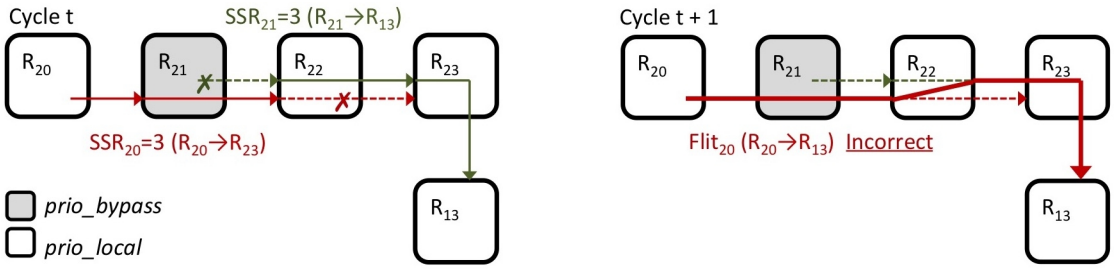
In both scenarios depicted in Fig. 3.7, significant throughput is loss due to false negatives. As we shall see in Section 3.4, by guaranteeing the avoidance of false negatives, our proposed solution ensures that the bypass paths shown in both solid red lines and partially dashed blue lines in Fig. 3.7(a) and Fig. 3.7(b) will be used, thus avoiding throughput loss. Further, besides false negatives due to a lack of knowledge regarding which SSRs have already lost SA-G earlier, false negatives can also occur when a higher priority SSR has to prematurely stop at an earlier intermediate router due to one of the blocking conditions described in Section 3.3.1 (e.g., a flit has to prematurely stop due to an existing buffered flit from the same source at an earlier intermediate router). Our proposed approach also guarantees the avoidance of all these types of false negatives as well.

3.3.3 Correctness issues with mixed-mode priorities

As noted in [45], all routers need to enforce the same priority for SSR arbitration, either *prio_local* or *prio_bypass*, to ensure correctness. If a mixed priority scheme is used, then it can lead to incorrect behavior due to the misrouting of flits. This can again happen because all routers are independently making their own allocation decisions at the same time, without knowing the allocation decisions that other upstream routers will make. Fig. 3.8(a) depicts an example where misrouting occurs due to the use of mixed-mode priorities. In this example, all routers except R_{21} employ the *prio_local* policy, but router R_{21} employs the *prio_bypass* policy. Router R_{20} sends an SSR to R_{23}, while at the same time, router R_{21} sends an SSR to R_{13}. At R_{21}, SSR_{20} wins output arbitration over SSR_{21} because of the *prio_bypass* policy at R_{21}, but SSR_{20} *loses* output arbitration to SSR_{21} at R_{22} and input arbitration to SSR_{21} at R_{23} because of the *prio_local* policy at R_{22} and R_{23}. Consequently, router R_{21} configures itself to bypass a flit from R_{20} to R_{22}, R_{22} configures itself to bypass a flit from R_{21} to R_{23}, and R_{23} configures itself to bypass a flit from R_{22} to R_{13}. This sets up an entire bypass path from R_{20} to R_{13}, which causes a *misrouting* of the flit from R_{20} to R_{13} instead of terminating at R_{23}, as requested by the corresponding SSR_{20}. This is depicted in Fig. 3.8(b) in the bold red line. As we shall see in Section 3.4.6, our proposed design works correctly under a mixed-mode priority setting.

3.4 SHARP NoC

In this section, we describe our proposed design named SHARP, for *Smart-Hop Arbitration Request Propagation*. Instead of arbitrating among $O(HPC_{max}^2)$ SSRs in



(a) Parallel SSR arbitration at cycle t.

(b) Misrouting of flit at cycle $t+1$.

Figure 3.8: Incorrect behavior in a mixed priority setting.

parallel at each input port, as SMART does, SHARP eliminates the quadratic arbitration problem by only considering at each input port the *winner* from the corresponding output port at the previous hop router. The previous hop router in turn only has to consider the winners from its previous hop routers that are adjacent to its input ports, and so on. We refer to this arbitration method as *propagation-based* SSR arbitration. As we shall see, besides eliminating the quadratic complexity of parallel-based SSR arbitration used by SMART, SHARP also guarantees the avoidance of false negatives. That is, SHARP obviates the two shortcomings of SMART, as outlined in Section 3.3. In the following, we further explore the properties of parallel SSR arbitration. We then apply the insights into the design of SHARP, and then we analyze the delays through the bypass path setups and flit traversals. Finally, we show how SHARP avoids false negatives.

3.4.1 Redundancies and Logical Dependencies

Consider again Fig. 3.4(a) where R_{23} receives up to 15 SSRs at its W_{in} port from sources within $HPC_{max} = 3$ hops away. Using this example, we observe two properties:

(1) *Redundancies*: We first observe that the set of SSRs considered at the W_{in}

port of R_{23} is a *subset* of SSRs considered at R_{22}, including local requests at R_{22}. In particular, the subset is *exactly* the set of SSRs that are (a) within $(HPC_{max} - 1)$ hops away from R_{22} and (b) go out through the E_{out} port of R_{22}, including the local request that won SA-L at E_{out} of R_{22}. For example, in Fig. 3.4(a), SSRs from R_{22}, R_{21}, R_{12}, R_{32}, R_{20}, R_{11}, R_{31}, R_{02}, and R_{42} (shown in light grey nodes) are considered by the input arbitration step at W_{in} of R_{23} to select a winner, as these SSRs are within $HPC_{max} = 3$ hops away. However, the same set of SSRs, including the local request from R_{22} itself, must also be considered at R_{22} to select a winner at its E_{out} port, as these SSRs are within $(HPC_{max} - 1)$ hops way from R_{22}.

We further observe that in parallel SSR arbitration, the input arbitration step at the W_{in} port of R_{23} will select the exactly *same* SSR as winner as the output arbitration step at the E_{out} port of R_{22}. The same observation can be made for any input port with the output port of the corresponding previous hop router (e.g., the winner at input port N_{in} of R_{22} is the same as the winner of the output port S_{out} of R_{12}). This means that the SSR arbitration step at each input port is *redundant* in the sense that the same arbitration logic and the same arbitration result are already required and available at the corresponding previous hop router. In turn, the output arbitration step at the previous hop router only needs to know about the winners at its input ports, but the arbitration logic and results for these input ports are already required and available at their corresponding previous hop routers as well, and so on. This redundancy imposes significant overhead in power, area and performance as the redundancy is replicated in parts at *every* router in the network. This is necessary because in parallel SSR arbitration, all routers independently make their own allocation decisions.

Alternatively, we can simply eliminate the SSR arbitration step (and the associated quadratic arbitration problem) all together at each input port by *propagating* the winner from the output arbitration step at the previous hop router. However, at each input port, we still have to perform the *blocking tests*, as described in Section 3.3.1, to determine if the propagated winning SSR needs to terminate or prematurely stop at the current input port.

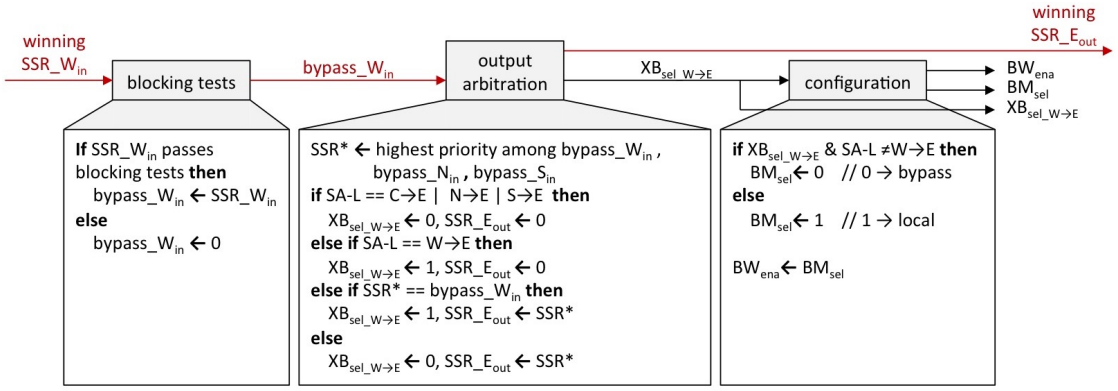
(2) *Logical Dependencies*: We next observe that in both the *prio_local* and *prio_bypass* policies, the priorities of SSRs are *distance-based*. In the *prio_local* case, an SSR 1-hop away can only be granted if there is no competing local request. Similarly, an SSR 2-hops away can only be granted if there are no competing SSRs from 1-hop away. In turn, an SSR k-hops away can only be granted if there are no competing SSRs from $(k-1)$-hops away, and so on. Therefore, the arbitration among SSRs that are k-hops away is *logically dependent* upon the arbitration result among SSRs that are $(k-1)$-hops away. This logical dependency explains the linearly increasing latencies observed in Fig. 3.6 with respect to HPC_{max} for the critical path through the parallel SSR arbitration logic implementations. Similarly, in the *prio_bypass* case, an SSR $(k-1)$-hops away can only be granted if there are no competing SSRs from k-hops away. This logical dependency also translates to the linearly increasing latencies observed in Fig. 3.6 with respect to HPC_{max}. As we shall see next, this logical dependency is already captured in our propagation-based arbitration approach in the sense that arbitration decisions made at a node $(k-1)$-hops away are logically dependent upon the winners selected at routers k-hops away. The important difference, however, is that the *redundant* logic is eliminated.

3.4.2 Propagation-based SSR Arbitration

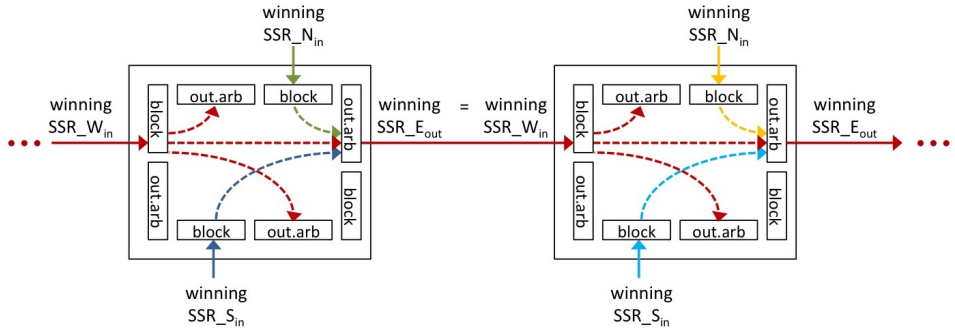
Fig. 3.9 depicts our *propagation-based* SSR arbitration approach in SHARP for SA-G. Instead of arbitrating among up to $HPC_{max}(2HPC_{max} - 1)$ SSRs in parallel at each input port, as SMART does, SHARP entirely eliminates the quadratic *SSR priority arbitration* function shown in Fig. 3.5 by exploiting the fact that the winning SSR at a given input port is the *same* as the winner from the corresponding output arbitration step at the previous hop router. Therefore, we can simply *propagate* that winner from the previous hop router. This winner is depicted as SSR_W_{in} in Fig. 3.9(a) and Fig. 3.9(b) at the input port W_{in}, which comes from the output port E_{out} at the previous hop router. At the input port, we now simply have to perform the *blocking tests* on SSR_W_{in}, as described in Section 3.3.1, to determine if a bypass path should be set up. The output arbitration step is nearly identical to SMART, except that we propagate the winning SSR to the output port E_{out} to the next hop router. Finally, the local configuration step is essentially the same as SMART. The key to SHARP is that the implementation complexity of all three steps, blocking tests, output arbitration, and local configuration, is independent of HPC_{max} and the number of routers in the network.

3.4.3 Implementation Details

In SMART, each dedicated SSR uniquely identifies the start router and the path (i.e., an XY or YX path) to the recipient router. The *hop_num* field indicates to the recipient router whether the SSR terminates at or before the recipient router, or passes through the recipient router. The uniqueness of each dedicated SSR, together with the *hop_num* field, also enables a recipient router to determine if the requested path makes



(a) Implementation of propagation-based SA-G.



(b) Propagation of the winning SSRs.

Figure 3.9: Implementation of propagation-based SA-G for *prio_local* in SHARP, focusing on W_{in} and E_{out}.

a turn or not at the current router, and if the turn should be to the left or right.

In SHARP, the wires used to carry *information* about the winning SSR to an input port do not uniquely identify a start router – the wires could carry information about *any* SSR within HPC_{max} hops away from the recipient router to its input port. Instead of a *hop_num* field, SSRs in SHARP are encoded by a *stop_id* field and a *distance* field. The *stop_id* in (x, y) coordinates is the location of the stop router where the SSR wishes to terminate. This information is sufficient for a recipient router to determine whether the SSR terminates at or before the recipient router, or passes through the recipient router (e.g., the recipient router can test if the (x, y) coordinates of the *stop_id* is the same as its own (x, y) coordinates). This information is also sufficient for a recipient router to determine if the requested path makes a turn or not at the current router, and if the turn should be to the left or right (e.g., if the (x, y) coordinates of the *stop_id* and the $(\hat{x}, y)$ coordinates of the recipient router are such that $\hat{x} < x$ and the y-coordinates are the same, then a left turn should be made)[7].

The *distance* field, which gets incremented at each hop, is used by the output arbitration step in SHARP to determine which of the remote SSRs from the input ports at the recipient router has the highest priority. For a tie-breaker, the *direction*-based priority can simply be determined by the arrival input port of the corresponding SSR (i.e., for E_{out}, the priority straight $>$ left-turn $>$ right-turn is equivalent to SSR_W_{in} $>$ SSR_N_{in} $>$ SSR_S_{in}).

Aside from a different way of implementing the SA-G step, as depicted in Fig. 3.9,

[7]The incoming SSR is assume to travel in the same direction until it reaches the *stop_id's* coordinate in that direction. i.e., if an SSR comes into W_{in} and does not terminate at the current router, then it is assumed to continue in the X direction out to E_{out} until the y-coordinate of the current router matches that y-coordinate of the *stop_id* (i.e., just the y-coordinate needs to be checked).

and a different way to encode the SSRs, as explained above, SHARP follows essentially the same pipeline structure as SMART, as depicted in Fig. 3.3, in which the following occurs:

- At cycle $t-1$, each router chooses a local candidate for every input/output port (SA-L) and prepares the corresponding SSR fields.
- At cycle t, an SSR is sent for every SA-L winner, and these SSRs are arbitrated using a propagation-based SA-G, as depicted in Fig. 3.9, in accordance to a priority scheme. The local configurations at each hop along the way are setup in the same cycle to establish the bypass datapaths.
- At cycle $t+1$, flits travel through the established bypass datapaths in a single cycle.

3.4.4 HPC_{max} Analysis

In this section, we analyze the maximum number of hops (HPC_{max}) that can be traversed within a single cycle using SHARP. To compare with SMART, we analyze delays using the same 45nm technology node used in the analysis for SMART [45]. Using 1mm per hop and a 1ns clock, [45] showed that an HPC_{max} of 13 or more hops could be achieved for repeated wires at 45nm. However, as discussed in Section 3.2 and as shown in Fig. 3.2, the delay at each hop includes not just the repeater and link segment delay, but also the delay through the bypass and crossbar muxes (as set by BM_{sel} and XB_{sel}, respectively). Fig. 3.10(a) depicts this delay through the datapath. In particular, the delay through the datapath for traversing up to HPC hops is given by Eq. (3.1), where t_{bypass} is the mux delays through the bypass and crossbar muxes, and t_{ℓ} is the

link segment delay per hop.

$$T_{data} = HPC \cdot (t_{bypass} + t_{\ell}) \tag{3.1}$$

The added mux delays per hop reduces HPC_{max} to 11 at 1 GHz, as explained in [45].

However, as shown in [45], the achievable HPC_{max} is actually limited by the delay through the SMART parallel SSR arbitration control path. This is depicted in Fig. 3.10(b). The delay through the SMART control path includes the link delays for the farthest SSRs to reach the recipent router, plus the delay through a parallel SA-G stage that must consider up to $HPC_{max}(2HPC_{max} - 1)$ SSRs. This delay for HPC hops is given by Eq. (3.2), where t_{ℓ} is again the link segment delay per hop for carrying the SSRs, and $t_{sa\text{-}g}^{smart}$ is the delay through a parallel SA-G stage.

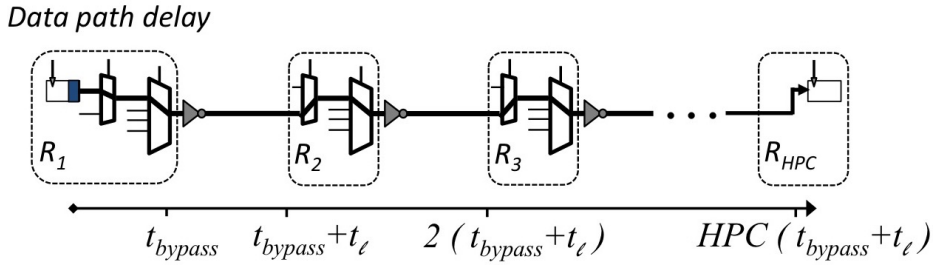
$$T_{cntrl}^{smart} = HPC \cdot t_{\ell} + t_{sa\text{-}g}^{smart} \tag{3.2}$$

As shown in [45], $HPC_{max} = 8$ is the optimal achievable HPC_{max} at 1 GHz.

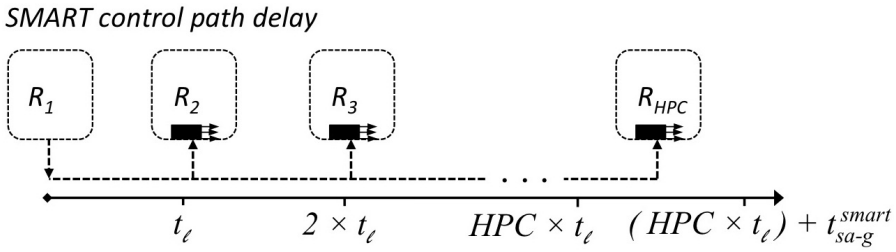
For the SHARP NoC design, the achievable HPC_{max} is also limited by the SA-G control path. This is depicted in Fig. 3.10(c). The delay through the SHARP control path includes the link delay at each hop, plus the SSR arbitration delay at each hop. This delay for HPC hops is given by Eq. (3.3), where t_{ℓ} is again the link segment delay per hop for propagating the winner SSR from one hop to the next, and $t_{sa\text{-}g}^{sharp}$ is the SSR arbitration delay per hop for each propagation-based SA-G stage.

$$T_{cntrl}^{sharp} = HPC \cdot (t_{\ell} + t_{sa\text{-}g}^{sharp}) \tag{3.3}$$

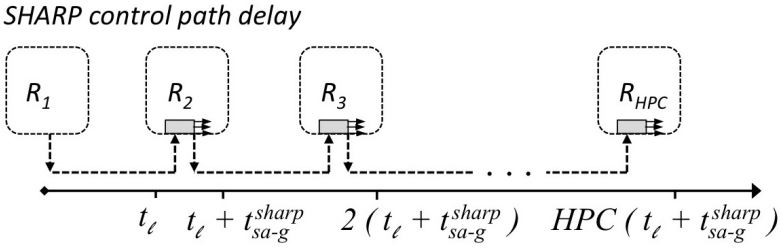
To derive the achievable HPC_{max} for SHARP, we use the same repeated wire configuration as [45] to obtain t_{ℓ}, and we use the Synopsys Design Compiler to synthesize our



(a) Single-cycle multi-hop datapath delay.



(b) SMART parallel SSR arbitration control path delay.



(c) SHARP propagation-based SSR arbitration control path delay.

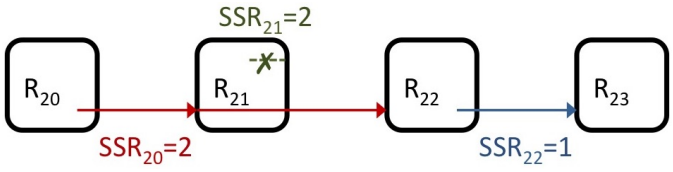
Figure 3.10: HPC_{max} analysis for SMART and SHARP.

propagation-based SA-G logic in a TSMC 45nm technology to obtain $t_{sa\text{-}g}^{sharp}$. Our synthesis results show that an $HPC_{max} = 6$ can be achieved. The total contribution from $HPC \cdot t_{\ell}$ is 414ps, and the total contribution from $HPC \cdot t_{sa\text{-}g}^{sharp}$ is 534ps, which together is under 1ns. Although our SSR arbitration logic is considerably simpler than SMART's, we nonetheless have a longer delay through the control path as the delay through our SSR arbiter has to be multiplied HPC times. As we shall see in Section 3.6, even though we have a smaller HPC_{max}, SHARP actually performs better than SMART for two reasons: (1) SHARP avoids false negatives, which leads to higher throughput. (2) With increasing traffic, long multi-hop paths become increasingly difficult to set up if the *prio_local* policy is used. This is because multi-hop paths often have to prematurely stop to give way to competing local requests. Therefore, a higher HPC_{max} value is not always helpful.

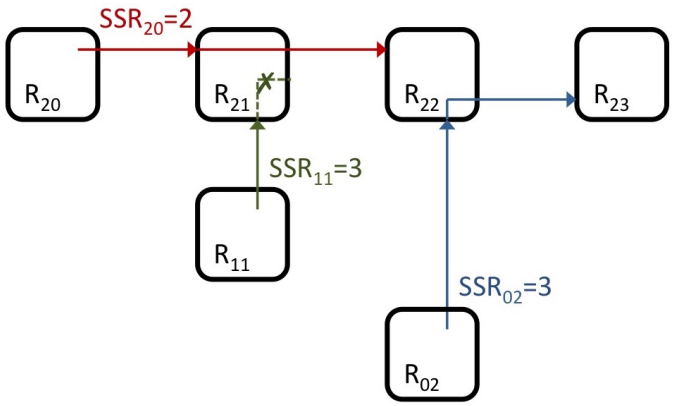
3.4.5 Avoidance of False Negatives

As discussed in Section 3.3.2, a shortcoming of SMART is that routers independently make their own allocation decisions without knowledge of allocation decisions made by upstream routers. For example, an SSR may no longer be active either because it lost SSR arbitration earlier or had to prematurely stop due to the failure of a blocking test. This leads to false negatives that cause throughput loss. In contrast, SHARP only propagates an SSR to the next hop only if it passes all blocking tests and *wins* SSR arbitration. Therefore, false negatives cannot occur (or are guaranteed to be avoided).

Fig. 3.11 shows the same two examples used to illustrate the false negative problem in Fig. 3.7 of Section 3.3.2. In particular, Fig. 3.11(a) illustrates the avoidance of



(a) Under *prio_bypass*.



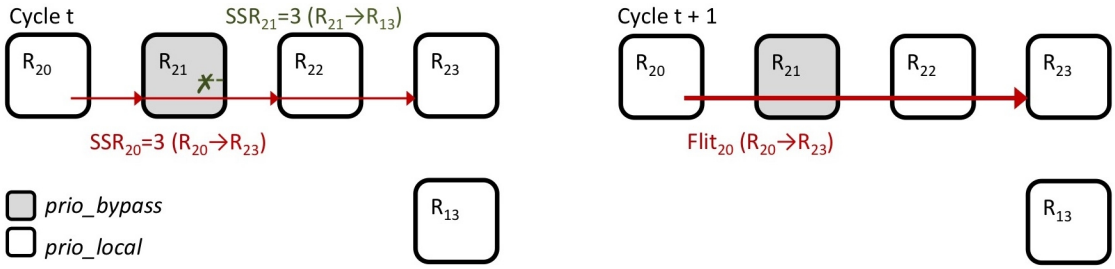
(b) Under *prio_local*.

Figure 3.11: SHARP avoids false negatives.

false negatives in the *prio_bypass* case. Because SSR_{20} wins output arbitration over SSR_{21} at R_{21}, only the winning SSR_{20} gets propagated to R_{22} where it terminates. At R_{22}, SSR_{22} is the only request competing for the output port to R_{23}, so it wins by default, thereby allowing R_{22} to send a flit to R_{23}. That is, SSR_{21} does not prevent SSR_{22} from winning as it does in the case of SMART. Therefore, both R_{20} and R_{22} can send their flits all the way to their respective stop routers, as shown in the solid red and blue lines in Fig. 3.11(a), resulting in higher throughput.

Similarly, Fig. 3.11(b) illustrates the avoidance of false negatives in the *prio_local* case. Because SSR_{20} wins output arbitration over SSR_{11} at R_{21}, only the winning SSR_{20} gets propagated to R_{22} where it terminates. At R_{22}, SSR_{02} is the only request competing for the output port to R_{23}, and therefore wins by default. That is, SSR_{21} does not prevent SSR_{02} from winning as it does in the case of SMART, which allows R_{02} to send a flit all the way to R_{23}. Therefore, both R_{20} and R_{22} can send their flits all the way to their respective stop routers, as shown in the solid red and blue lines in Fig. 3.11(b), which again results in higher throughput.

As we shall see in Section 3.6, the avoidance of false negatives is crucial for throughput at higher traffic loads when contentions among competing SSRs are fierce. Also, as we shall see in our evaluations, the avoidance of false negatives is crucial for setting up longer multi-hop paths at higher traffic loads that would otherwise be prematurely stopped. For example, in Fig. 3.11(b), SSR_{02} would have prematurely stopped in SMART at R_{22}.



(a) SSR arbitration in SHARP at cycle t.

(b) Correct flit traversal at cycle $t+1$.

Figure 3.12: SHARP ensures correctness under a mixed priority setting.

3.4.6 Ensuring correctness under mixed-mode priorities

As discussed in Section 3.3.3, another shortcoming of SMART is that routers cannot mix priorities because doing so can lead to incorrect behavior. In contrast, routers in SHARP are free to arbitrarily mix priority policies (*prio_local* or *prio_bypass*) without leading to incorrect behavior. Fig. 3.12 depicts the same example used to illustrate the correctness problem in Fig. 3.8 of Section 3.3.3. In this example, because SSR_{20} wins output arbitration over SSR_{21} at R_{21}, only the winning SSR_{20} gets propagated down to R_{22} and R_{23} where SSR_{20} terminates, as shown in Fig. 3.12(a). This sets up a bypass path from R_{20} to just R_{23}. Then in cycle $t+1$, the flit from R_{20} correctly routes directly to R_{23}, as shown in the bold red line in Fig. 3.12(b).

3.5 Delivery of Companion SSR Signals

As described in Section 3.3, the SA-G process in SMART comprises four parts: (1) SSR priority arbitration, (2) blocking tests, (3) output arbitration, and (4) bypass configurations. For SSR priority arbitration, SMART only needs the *hop_num* informa-

Table 3.1: Companion SSR signals.

Signal	Width (in bits)	Description
source_id	$2 \times \lceil \log_2(HPC_{max}) \rceil$	Specifies the ID of the source router where the SMART-hop is initiated.
vnet_id	$\lceil \log_2(N_{vnet}) \rceil$	Specifies the ID of the virtual network to which the flit belongs.
head_flag	1	Indicates that the flit is a head flit.
eject_flag	1	Indicates that the flit should be ejected.
eject_port_id	$\lceil \log_2(N_{port}) \rceil$	Specifies the ID of the ejection port.

tion to determine which remote SSR has the highest priority and therefore should win the arbitration. However, to perform the blocking tests on the winning SSR, as described in Section 3.3.1, additional *companion* SSR signals are required, as shown in Table 3.1, where N_{vnet} is the number of virtual networks (e.g., $N_{vnet} = 3$), and N_{port} is the number of router ports (e.g., $N_{port} = 5$). These companion SSR signals are used in the blocking tests in the following manner:

- SMART uses a VC allocation table to keep track of the allocations of VCs. The VC allocation is identified by a combination of *source_id* and *vnet_id*. To check if there is a *prematurely* stopped flit among its buffered flits from the same source as the winning SSR, the VC allocation table is consulted using (*source_id*, *vnet_id*) as the key. If a VC has already been allocated to this source, then the bypassing flit will be stopped at this VC to avoid flit re-ordering.

- If the corresponding flit is a head flit, as indicated by *head_flag*, then the immediate downstream router is checked for a free VC to allocate to this head flit (for the corresponding virtual network as specified by *vnet_id*).

- To check if the current router is the final destination for a given SSR, *eject_flag* is checked. If this flag indicates ejection, then the corresponding flit will retire directly to the network interface buffer, as specified by *eject_port_id*, bypassing the destination router's pipeline.

As observed in [22], the articles that describe SMART [21, 45, 46] lack any mention or analysis of these companion SSR signals, but they are needed for the blocking tests. In turn, these blocking tests are needed before output arbitration and bypass configurations can be performed. In the absence of any analysis in the SMART articles, [22] assumes that these companion SSR fields are *broadcast* with each SSR together with the *hop_num* field. [22] argues that this is very expensive in terms of wiring cost because of two reasons: (1) the combined width of these companion SSR fields is much wider than just the *hop_num* field alone; (2) all these companion SSR fields must be included with each of the $HPC_{max}(2HPC_{max}-1)$ SSRs that each input receives from routers HPC_{max} hops away for SSR priority arbitration.

Instead, [22] proposes to split the SA-G process into two pipeline stages: a *pre-SSR* stage and a *post-SSR* stage[8]. We refer to this design as *SSR-Net* [22]. In particular, in SSR-Net, their pre-SSR stage performs the SSR priority arbitration step among $HPC_{max}(2HPC_{max}-1)$ competing SSRs at each input. This stage is identical to the

[8]In [22], the first stage is called a pre-SSR stage, as we also call it in this work, but [22] refers to the second stage as SA-G, which we find confusing since both stages are part of the overall SA-G process. Therefore, in this work, we instead refer to the second stage of the SA-G process as the post-SSR stage.

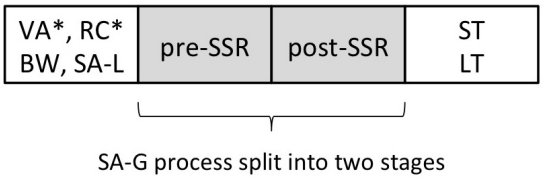


Figure 3.13: SSR-Net pipeline. SSR-Net implements SA-G in two stages. SSR priority arbitration is performed in the *pre-SSR* stage using a parallel-based SSR architecture scheme, and the delivery of the companion SSR signals and the bypass configurations are performed in the *post-SSR* stage. * indicates head-flit only.

parallel SSR priority arbitration step in SMART, as explained in Section 3.3.1, which only requires the *hop_num* field be broadcasted. Like SMART, each router in SSR-Net independently makes its own SSR arbitration decision, without knowledge of arbitration decisions made by upstream routers. As such, SSR-Net suffers from the same shortcomings as SMART, as described in Section 3.3. Specifically, SSR-Net suffers from the same quadratic complexity and false negative problems that afflict SMART. However, SSR-Net improves significantly in wiring cost over SMART because it does not broadcast the companion SSR signals with each SSR. Instead, SSR-Net uses an auxiliary network (called an SSR network in [22]) to propagate the companion SSR signals that correspond to the winning SSRs from the pre-SSR stage. This propagation of the companion SSR signals happens in a separate *post-SSR* stage, in which the remaining steps of blocking tests, output arbitration, and bypass configurations are also performed. Therefore, a flit needs to spend at least three cycles at the source router before making a SMART-hop traversal, one more cycle than needed by SMART or SHARP. The SSR-Net router pipeline is shown in Fig. 3.13. In summary, SSR-Net takes the following steps to setup

a SMART-hop:

- At cycle $t-1$, all source routers perform SA-L to choose a local winner for each input/output port.
- At cycle t, instead of broadcasting all the fields in an SSR, the source routers just broadcast their *hop_num* field to recipient routers HPC_{max} hops away. The recipient routers in turn perform SSR priority arbitration on the received SSRs in their *pre-SSR* stage.
- At cycle $t+1$, source routers send the remaining companion SSR fields through the auxiliary SSR network in their *post-SSR* stage. Every router then receives the remaining companion SSR fields just for the winning SSRs at each of its inputs and configures itself appropriately for the subsequent single-cycle multi-hop traversal.
- At cycle $t+2$, flits that won both SA-L and SA-G stages at source routers proceed with their SMART-hop traversals.

Although SSR-Net performs SA-G in two pipeline stages, both SMART and SSR-Net will essentially configure the same bypasses, including the same false negatives. However, as we shall see in Section 3.6, the extra pipeline stage causes noticeably higher latencies in our evaluations, especially under low traffic loads.

In SHARP, rather than receiving up to $HPC_{max}(2HPC_{max}-1)$ SSRs in parallel at each input, as SMART and SSR-Net do, SHARP simply receives one winning SSR that corresponds to the winner from the corresponding output arbitration step at the previous hop router. The propagation of the winning SSRs also includes the propagation of the corresponding companion SSR signals. Thus, like SSR-Net, SHARP avoids the

Table 3.2: How control (from SSRs) and data signals are transmitted.

Category	Fields	SMART	SSR-Net	SHARP
SSR arbitration	hop_num	broadcast	broadcast	
	stop_id			propagated
	distance			propagated
Companion SSRs	source_id	broadcast	propagated	propagated
	vnet_id	broadcast	propagated	propagated
	head_flag	broadcast	propagated	propagated
	eject_flag	broadcast	propagated	propagated
	eject_port_id	broadcast	propagated	propagated
Data	flit_data	propagated	propagated	propagated
	flit_metadata	propagated	propagated	propagated
	VC_control	propagated	propagated	propagated

broadcasting of the companion SSR signals from a quadratic number of SSRs from HPC_{max} hops away. SHARP further reduces wiring cost because it also avoids the broadcasting of the control signals that are needed for SSR priority arbitration from a quadratic number of SSRs from HPC_{max} hops away, as SMART and SSR-Net require (i.e., the *hop_num* field). SHARP simply propagates the control signals (i.e., the *stop_id* and *distance* fields) of the winning SSRs from one router to the next.

Table 3.2 summarizes how the different SSR control signals are transmitted in SMART, SSR-Net, and SHARP. For completeness, Table 3.2 also summarizes data sig-

nals (corresponding to the routing of flits) that are propagated from one router to the next.

3.6 Evaluation

3.6.1 Experimental Setup

For our evaluations, we consider an 8×8 mesh network with 64 cores. Each core comprises a processor node, an L1 data/instruction cache, a slice of a shared L2 cache, and a slice of the cache coherency directory. The cores operate on a 1 GHz clock. We allocate a 32KB private 4-way L1 cache to each of the 64 cores, and we allocate a 1MB shared 8-way L2 cache. For cache coherence, we employ a MOESI directory protocol [32]. We assume each core occupies a 1mm×1mm area with 1mm SMART-hop links between them. For SMART, SSR-Net, and our SHARP approach, we consider both XY and YX routing paths. Therefore, for SMART and SSR-Net, SSR dedicated links can stretch along both XY and YX directions. For our evaluations, we consider $HPC_{max} = 8$ for SMART and SSR-Net, which is the best achievable HPC_{max} for a 2D configuration [45]. For SHARP, we consider $HPC_{max} = 6$, which is the best achievable HPC_{max} for SHARP under the same system configuration.

For all simulations, we set #VCs to 12 so that are enough VCs to prevent input buffer contentions for the network size and HPC_{max} values evaluated. The packet size for synthetic benchmark evaluations is fixed to 1 flit. A head flit goes through a three or four stage pipeline in the SMART and SSR-Net approaches, respectively, as shown in Figs. 3.3 and 3.13. For the memory subsystem, we use a shared L2 cache because of its

higher injection rate over a private type. These system parameters are summarized in Table 3.3.

For performance evaluation, we use Garnet [7] for NoC simulation and gem5 [15] for full-system simiulation. We use DSENT [67] integrated into Garnet to calculate power and area results. The networks are evaluated by running both synthetic and real application benchmarks. For synthetic traffic, we use three traffic models: Uniform, Tornado, and Bitcomp (bit-complement). Under Uniform traffic, each core will generate traffic to a destination chosen at uniform random. Under Tornado traffic, a core at location (x, y) will generate traffic to a destination at location (dx, y), where $dx = (x+3)$ mod k, where k is the radix of the mesh network (e.g., $k = 8$). Under Bitcomp traffic, a core at location (x, y) will generate traffic to a destination at location (dx, dy), where dx and dy are bitwise complements of the binary encodings for x and y, respectively (e.g., if $x = 000$, then $dx = 111$). For real-case scenarios, we use selected applications from the PARSEC and SPLASH-2 benchmark suites and ran them through the gem5 full-system simulator. These benchmarks are also summarized in Table 3.3.

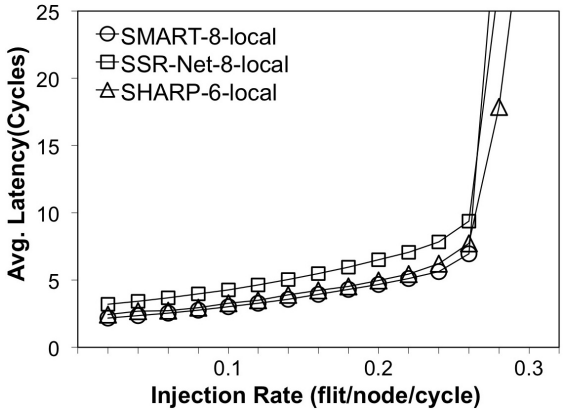
To name a design, we use the following template: [approach]-[HPC_{max}]-[priority], in which, an approach can be SMART, SSR-Net, or SHARP, and a priority scheme can either be *local* or *bypass* (e.g. SMART-8-bypass).

3.6.2 Performance Comparisons

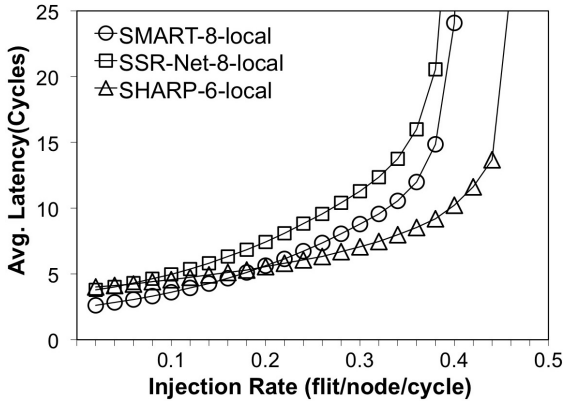
We first compare the performance of SMART, SSR-Net, and SHARP for average flit latency and link utilization under synthetic traffic patterns.

Table 3.3: System parameters and benchmarks.

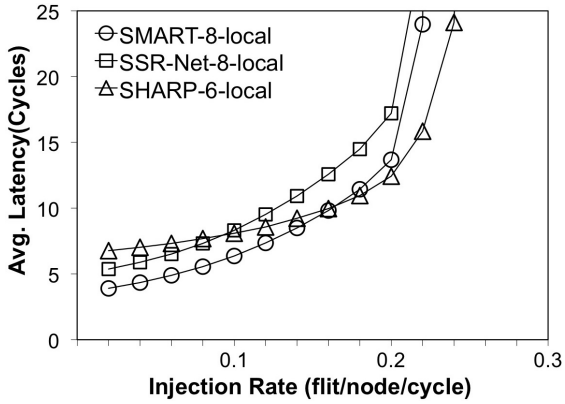
Processor	
Core	Freq.: 1 GHz; #Cores: 64; Area: 1mm×1mm; Tech.: 45nm
Caches	L1: 64 private 4-way, 32KB/core; L2: 64 shared, 8-way, 1MB; Coherence: MOESI(blocking)
NoC	
Router	Freq.: 1GHz; Virtual Networks: 3; Virtual Channels: 12; Routing: XY/YX; Flits/packet: 1 (control), 5 (data)
Interconnect	Topology: 8×8 mesh; Link: 1mm; Width (flit): 128-bit
Applications	
Synthetic	uniform, tornado, bit-compliment
Applications	fft, barnes(br), lu_cb(lc), lu_ncb(lnc), radix(rd), bodytrack(bt), cholesky(ck), facesim(fs), blackscholes(bs), swaptions(sw), water_nsquared(wns), radiosity(rs), raytrace(rt)



(a) Tornado Traffic.

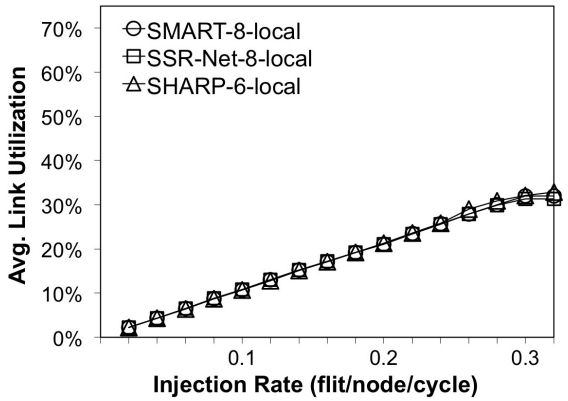


(b) Uniform Traffic.

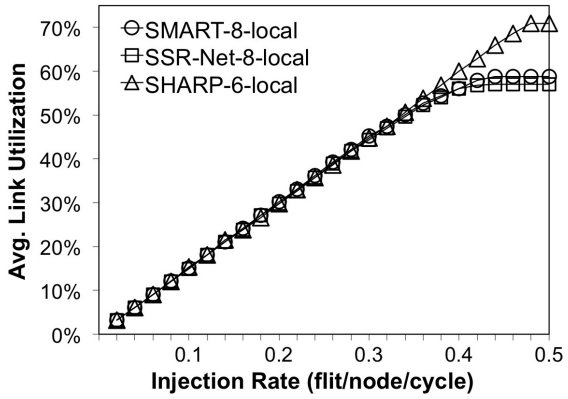


(c) Bitcomp Traffic.

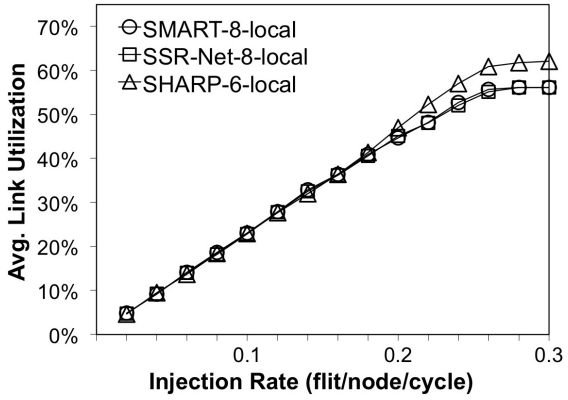
Figure 3.14: Average network latency for *prio_local* policy. SHARP achieves better network latencies due to the guaranteed avoidance of false negatives.



(a) Tornado Traffic.

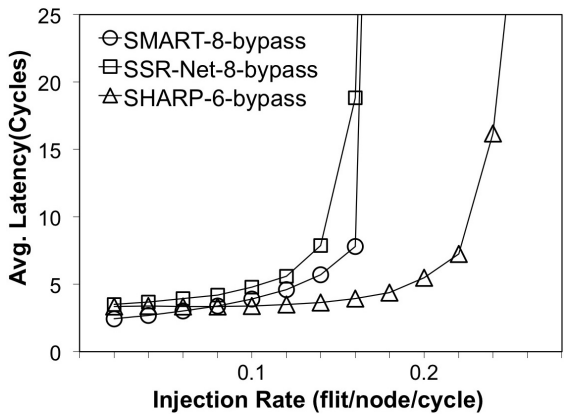


(b) Uniform Traffic.

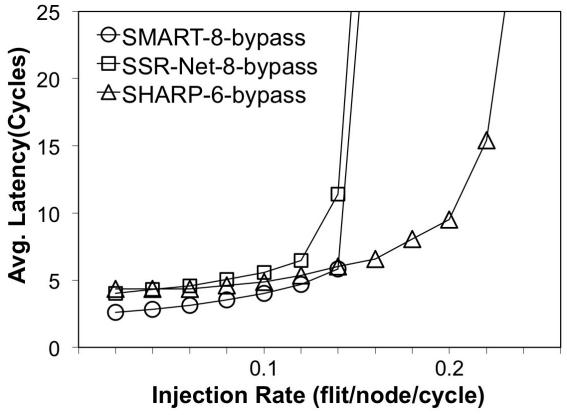


(c) Bitcomp Traffic.

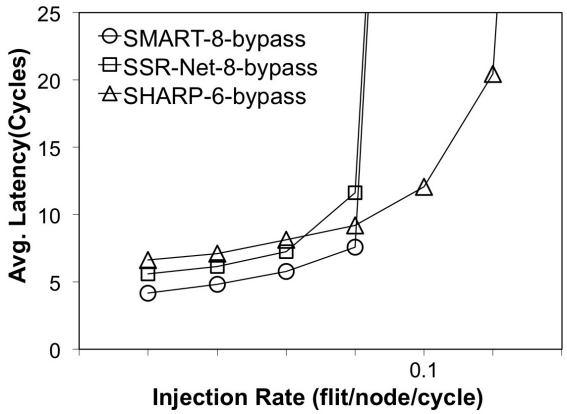
Figure 3.15: Average link utilization for *prio_local* policy. SHARP achieves higher link utilizations due to the guaranteed avoidance of false negatives.



(a) Tornado Traffic.

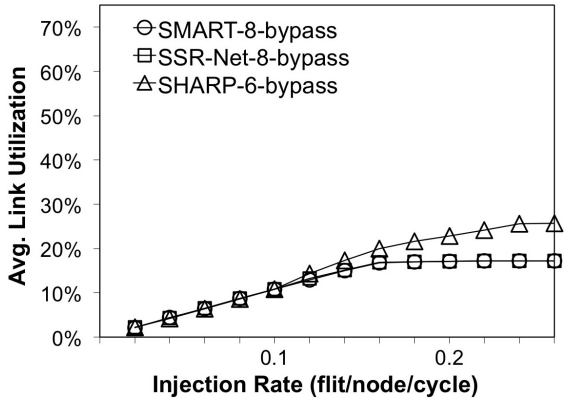


(b) Uniform Traffic.

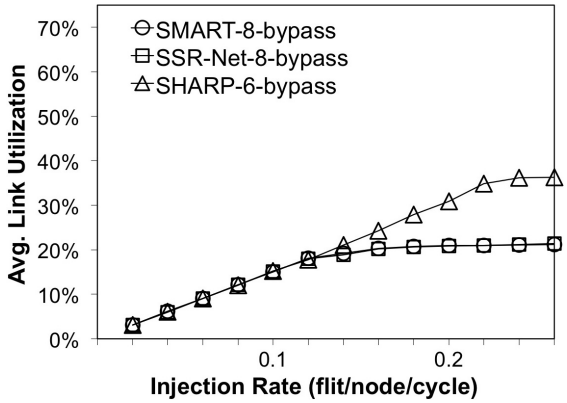


(c) Bitcomp Traffic.

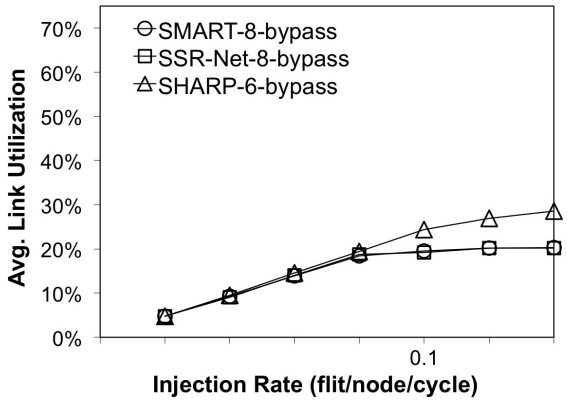
Figure 3.16: Average network latency for *prio_bypass* policy. SHARP achieves better network latencies due to the guaranteed avoidance of false negatives.



(a) Tornado Traffic.

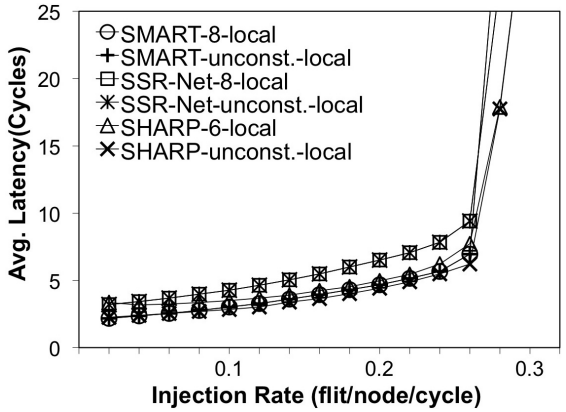


(b) Uniform Traffic.

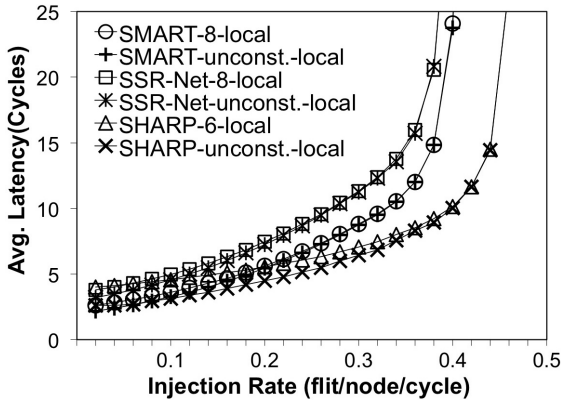


(c) Bitcomp Traffic.

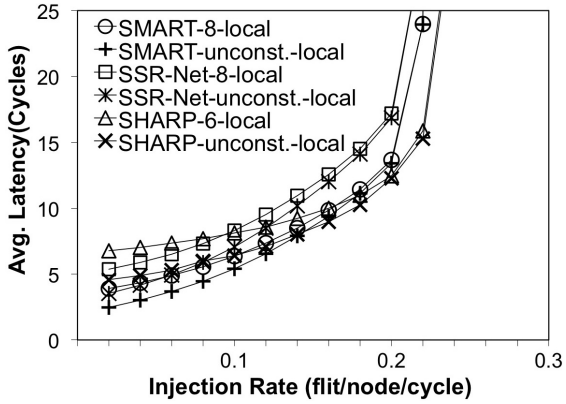
Figure 3.17: Average link utilization for *prio_bypass* policy. SHARP achieves higher link utilizations due to the guaranteed avoidance of false negatives.



(a) Tornado Traffic.

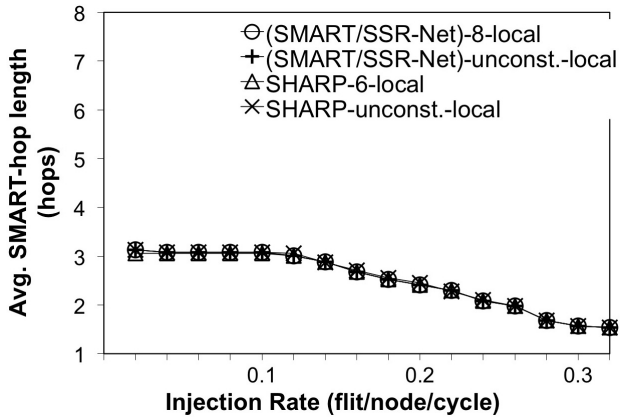


(b) Uniform Traffic.

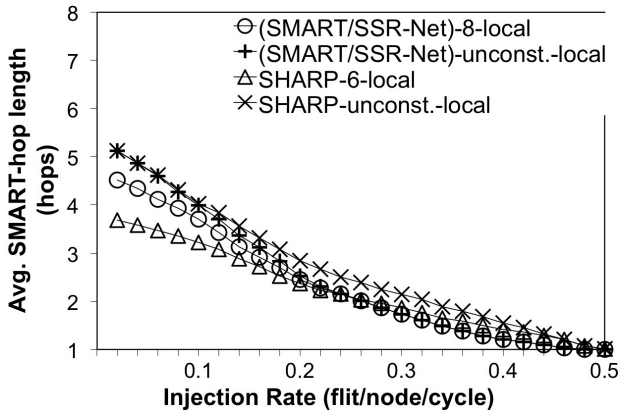


(c) Bitcomp Traffic.

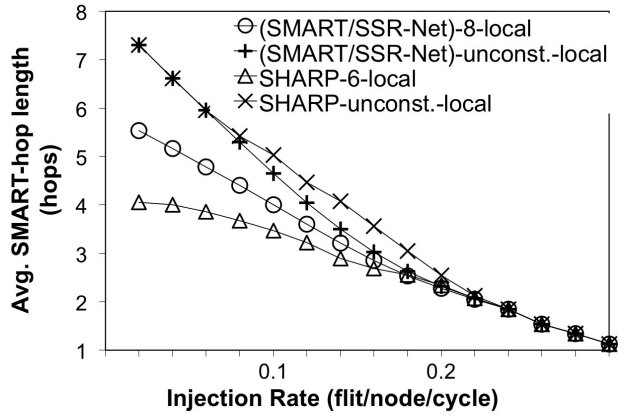
Figure 3.18: Impact of HPC_{max} on performance for *prio_local* policy.



(a) Tornado Traffic.

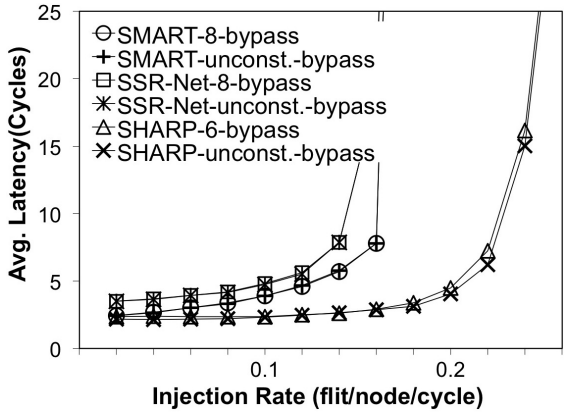


(b) Uniform Traffic.

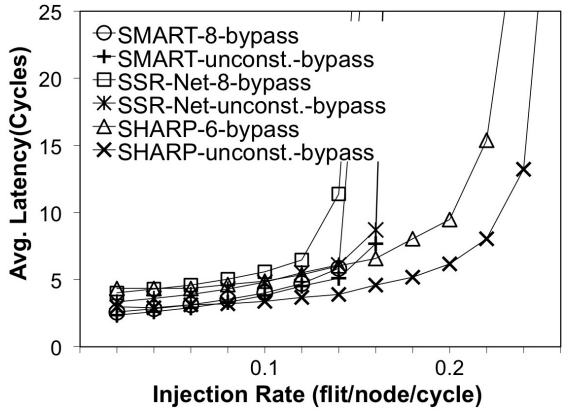


(c) Bitcomp Traffic.

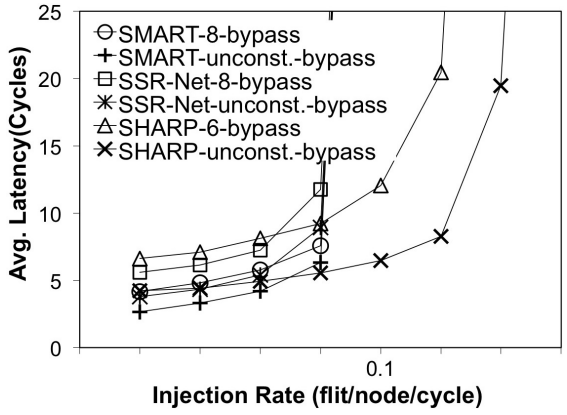
Figure 3.19: Average SMART-hop length for *prio_local* policy.



(a) Tornado Traffic.

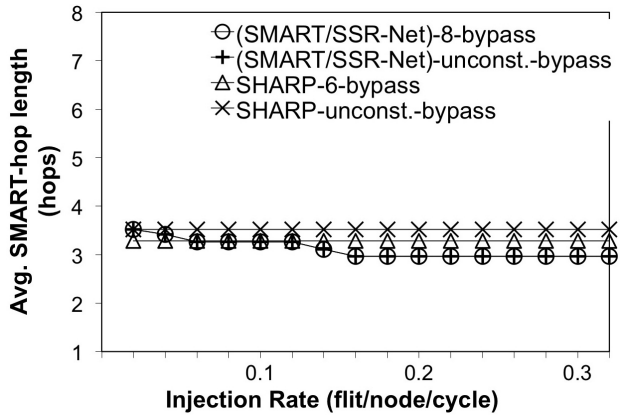


(b) Uniform Traffic.

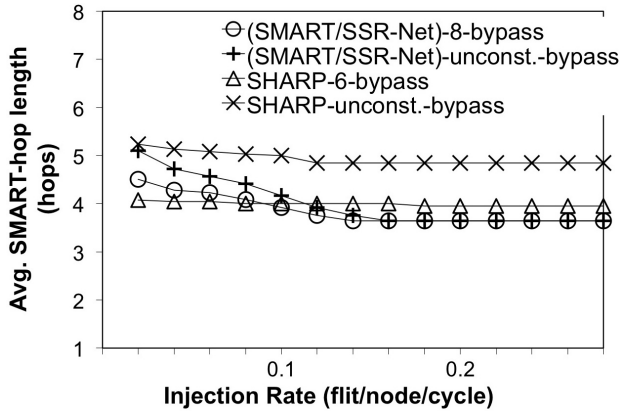


(c) Bitcomp Traffic.

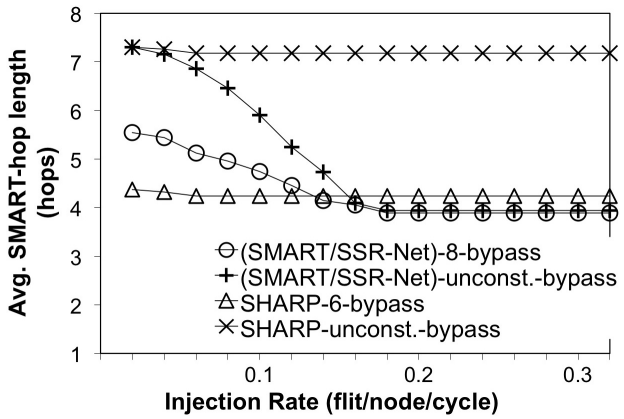
Figure 3.20: Impact of HPC_{max} on performance for *prio_bypass* policy.



(a) Tornado Traffic.



(b) Uniform Traffic.



(c) Bitcomp Traffic.

Figure 3.21: Average SMART-hop length for *prio_bypass* policy.

Local scheme: Figs. 3.14 and 3.15 show the results for the *prio_local* scheme. For the latency results shown in Figs. 3.14(a) to 3.14(c), we see that SHARP has slightly higher latencies for low injection rates, but much better latencies for moderate to high injection rates, in comparison to SMART and SSR-Net. These observations show that the improvement in the network capacity is negligible for the Tornado, but is quite substantial for the Uniform and Bitcomp traffic patterns. The results show 16% and 10% improvement in throughput over SMART and SSR-Net for Uniform and Bitcomp, respectively.

Figs. 3.15(a) to 3.15(c) show the link utilization results, where the average link utilization corresponds to the average number of network links being used per cycle (i.e., with a flit traversing it). The results in Figs. 3.15(a) to 3.15(c) show that the link utilization is almost identical for low injection rates, but SHARP significantly outperforms SMART and SSR-Net in link utilization for moderate to high injection rates. These network latency and link utilization results, which are correlated, clearly demonstrate the inefficiencies that SMART and SSR-Net face due to the *false negatives* problem described in Section 3.3. It is worth noting that both SMART and SSR-Net have similar network latencies and link utilizations because both employ the same parallel SSR arbitration mechanism. The difference in network latencies between SMART vs. SSR-Net is due to the extra pipeline stage required by the SSR-Net approach. For SHARP, SSR arbitration is performed by propagating the highest priority SSR to the next router. Therefore, the avoidance of false negatives is guaranteed, as explained in Section 3.4.

Bypass scheme: Figs. 3.16 and 3.17 show the results for the *prio_bypass* scheme. For the latency results shown in Figs. 3.16(a) to 3.16(c), we see a similar trend as the re-

sults for the *prio_local* policy, where SHARP has slightly higher latencies for low injection rates, but much better latencies for moderate to high injection rates, in comparison to SMART and SSR-Net. These observations show that the improvement in network capacity is quite substantial for all three traffic patterns. The results show 50% improvement in throughput for Tornado and Uniform traffic and 57% improvement in throughput for Bitcomp traffic, in comparison to SMART and SSR-Net.

For the link utilization results shown in Figs. 3.17(a) to 3.17(c), we also see a similar trend as the results for the *prio_local* policy, where SHARP significantly outperforms SMART and SSR-Net in average link utilization. In comparison to the results for the *prio_local* policy, we see the improvements in network latency and link utilization are even more pronounced due to the higher impact of *false negatives* when using the *prio_bypass* policy, which severely affect the performance of SMART and SSR-Net. In particular, with the *prio_bypass* policy, the chances for false negatives are much higher. Again, SHARP avoids the problems associated with false negatives by avoiding them all together.

3.6.3 Sensitivity Analysis

Next, we study the impact of HPC_{max} on the three designs (i.e., SMART, SSR-Net, and SHARP). To perform this evaluation, we consider *unconstrained* HPC_{max} configurations in which SMART-hop paths can be setup from any router to any router along either XY or YX paths[9]. The goal of this experiment is to evaluate the impact that a higher HPC_{max} has on performance. In particular, we compare average flit latency and average SMART-hop length results for both the *prio_local* and *prio_bypass* policies.

[9]For the proposed network size, $HPC_{max} = 14$ is the maximum possible value.

Local scheme: Figs. 3.18 and 3.19 show the impact of an unconstrained HPC_{max} under the *prio_local* policy. For the latency results depicted in Figs. 3.18(a) to 3.18(c), all three designs show small improvements under low injection rates when HPC_{max} is unconstrained. However, the advantage of unconstrained HPC_{max} quickly diminishes with increasing injection rates. This is because it becomes increasingly difficult to setup long SMART-hop paths under the *prio_local* policy since a SMART-hop path must *terminate* at a router if there is a competing SSR that either originates locally from this router or is *closer* to this router.

This phenomenon can be observed in Figs. 3.19(a) to 3.19(c). As shown in Figs. 3.19(a) to 3.19(c), the average SMART-hop length decreases with increasing injection rates for all three traffic patterns. For all three traffic patterns, the average SMART-hop length diminishes to below 3 for moderate injection rates and to 1 under high injection rates. For Tornado traffic, which represents short distance traffic patterns, all three designs show identical results. For Uniform and Bitcomp traffic that have a distribution of flows with longer distances, we see a higher divergence in the beginning when the injection rate is low where there is little or no contention among the flows. Although we see higher average SMART-hop lengths at low injection rates with an unconstrained HPC_{max}, the impact on network latency is limited.

Bypass scheme: Similarly, Figs. 3.20 and 3.21 show the impact of an unconstrained HPC_{max} under the *prio_bypass* policy. For Tornado traffic, since all traffic flows are at most 4 hops away to their destinations[10], which is less than the original HPC_{max} (i.e., 8 for SMART/SSR-Net, and 6 for SHARP), having an unconstrainted HPC_{max}

[10]For an 8×8 network, a router at location (x, y) will generate traffic for a destination at location (dx, y), where $dx = (x + 3) \mod 8$, which is at most 4 hops away.

does not improve performance. This can be observed in Fig. 3.20(a) and Fig. 3.21(a) where the average latency and SMART-hop length results are similar for the two configurations (i.e., the original vs. unconstrained HPC_{max}). For the Uniform and Bitcomp traffic patterns, we can also observe in Figs. 3.20(b) and 3.20(c) and Figs. 3.21(b) and 3.21(c) that having an unconstrainted HPC_{max} provides only modest improvements in average latency and SMART-hop length results for SMART and SSR-Net. This is due to the adverse effects of false negatives in the parallel SSR arbitration mechanisms used by these designs such that a higher HPC_{max} does not necessarily translate to better results. For Bitcomp traffic, although both SMART and SSR-Net have higher average SMART-hop lengths under low-injection rates for an unconstrainted HPC_{max}, the average SMART-hop lengths quickly converge to lower average lengths with increasing injection rates.

On the other hand, for Uniform and Bitcomp traffic, we can observe significant benefits when a higher HPC_{max} is used with SHARP (with an unconstrainted HPC_{max} being the best case). We observe in Figs. 3.21(a) to 3.21(c) that the average SMART-hop length remains high with increasing injection rates for the unconstrainted HPC_{max} case, which leads to lower network latencies as well as higher throughputs. This means that we can further improve the performance of SHARP by possibly employing specialized circuit techniques that can extend HPC_{max} by reducing the gate delays in the arbitration logic, whereas similar circuit optimization techniques that can extend HPC_{max} for SMART and SSR-Net may not be beneficial.

3.6.4 Wiring and Area Comparisons

In this section, we compare SHARP to SMART and SSR-Net in terms of wiring and area costs. We first compare the wiring costs of SHARP vs. SMART and SSR-Net. For all three designs, Table 3.1 specifies the *data* signals associated with the flit being forwarded and the *control* signals needed for SSR arbitration. For SSR-Net, we also implemented a *coarse* version with a single bit-width pre-SSR, as proposed in [22] for reducing wiring costs[11] to represent the best that the other techniques can achieve in terms of wiring cost. We label this coarse version of SSR-Net as SSR-Net©.

In Fig. 3.22(a), we compare the wiring costs for SMART, SSR-Net, and SHARP with respect to HPC_{max} = 1, 2, 4, 6, and 8. The wiring costs are normalized to the cost of the data signals to highlight the overhead of the control signals required for each of the three approaches. That is, we divide the wiring cost of the control signals by the wiring cost of the data signals, where the wiring costs for the data signals are the same for all three approaches, but the wiring costs for the control signals are considerably higher for SMART and SSR-Net. Recall that in SMART and SSR-Net, the parallel SSR arbitration approach needs to arbitrate among up to $HPC_{max}(2HPC_{max} - 1)$ SSRs per input port at each router from routers up to HPC_{max} hops away. Therefore, we expect to see a *quadratic* increase in wiring costs with increasing HPC_{max}. This quadratic increase can indeed be observed in Fig. 3.22(a) for SMART, SSR-Net, and SSR-Net©. As explained in Section 3.3[12], SSR-Net and SSR-Net© improve over SMART in wiring cost due to the use of an SSR proxy network to propagate companion SSR signals instead

[11] Using coarse pre-SSRs significantly exacerbates the *false negatives* problem and increases dynamic energy consumption, especially for large HPC_{max} values.

[12] Have to change the section citation once we have a companion SSR signal section.

of each router broadcasting all companion SSR signals in parallel to all routers HPC_{max} away. However, we can observe still a quadratic increase in wiring cost for SSR-Net and SSR-Net© due to the fact that each router still needs to broadcast in parallel the main SSR control signal (*hop_num*) to all routers that are HPC_{max} hops away. That is, each input port at each router still receives up to $HPC_{max}(2HPC_{max} - 1)$ SSRs to perform the parallel SSR arbitration step.

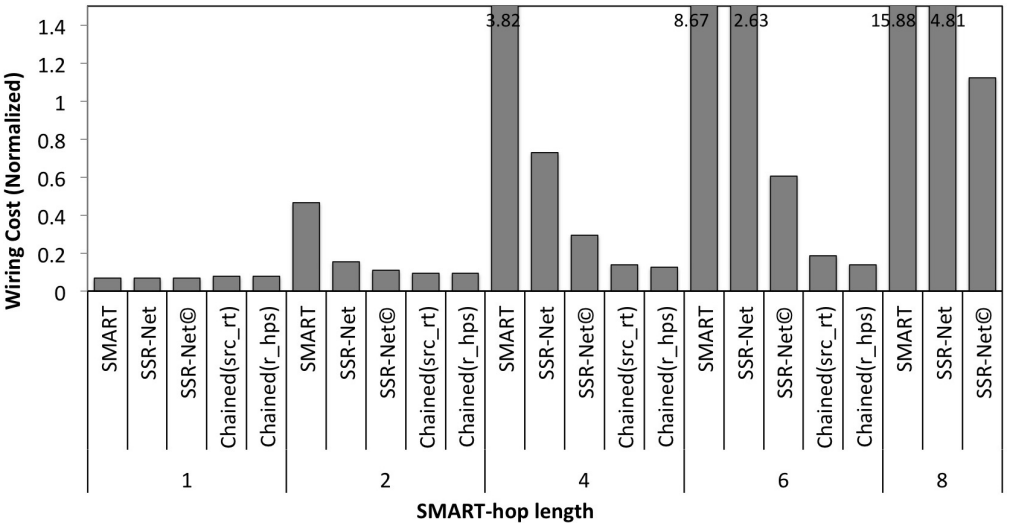
Recall that in SHARP, SSR arbitration is performed by propagating the highest priority SSR to the next router. Therefore, each router just needs to consider one SSR at each input port. Nonetheless, we can observe that the wiring cost for SHARP does increase, albeit slowly, with increasing HPC_{max}. This is due to the fact that the width of some SSR control signals (e.g., *hop_num*) are increasing logarithmically with respect to HPC_{max} (see Table 3.1). Indeed, as shown in Fig. 3.22(a), the wiring cost for SHARP increases very slowly with increasing HPC_{max}. For $HPC_{max} = 8$, the wiring overheads for the SSR control signals of SSR-Net and SMART are 4.81-15.88 times the wiring cost of data signals, which is to be expected due to the quadratic increase in wiring costs. Even in the optimized coarse version of SSR-Net, labeled as SSR-Net©, the wiring overhead for the SSR control signals exceeds the wiring cost of data signals by a factor of 1.12 times.

We next compare the area costs of the SSR control logic for SHARP vs. SMART and SSR-Net. In Fig. 3.22(b), we compare the area costs of the SSR control logic for all three approaches. We again compare the area costs for SHARP, SSR-Net, and SHARP with respect to $HPC_{max} = 1$, 2, 4, 6, and 8. The area costs shown in Fig. 3.22(b) are normalized to the SSR control logic cost with $HPC_{max} = 1$. As shown in Fig. 3.22(b),

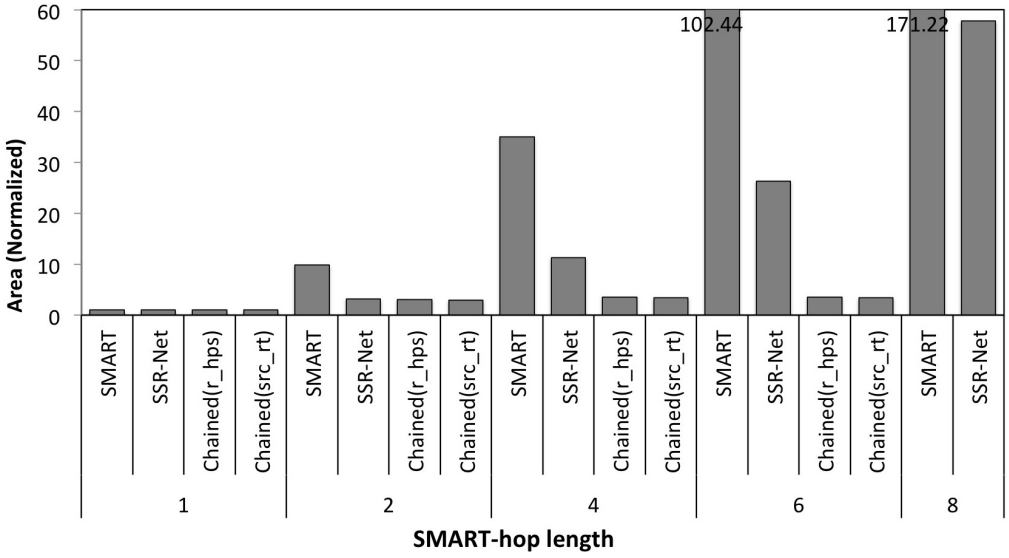
the SSR control logic cost is essentially the same for $HPC_{max} = 1$ for all three approaches since all three approaches consider exactly the same number of SSRs, one SSR per input port. Just as we can observe for the wiring costs, the area costs for the SSR control logic for SMART and SSR-Net are also increasing *quadratically* with respect to HPC_{max} since both SMART and SSR-Net have to consider a quadratically increasing number of SSRs with respect to HPC_{max}. On the other hand, with SHARP, we can observe that the area cost for the SSR control logic increases slowly with respect to HPC_{max}. This is again due to the fact that SHARP performs SSR arbitration by propagating the highest priority SSR to the next router. Therefore, each router just needs to consider one SSR at each input port. The increase in area observed in Fig. 3.22(b) is again due to the fact that the width of some SSR control signals (e.g., *hop_num*) are increasing logarithmically with respect to HPC_{max} (see Table 3.1), which leads to an increase in the SSR control logic area. For $HPC_{max} = 8$, the area overheads of SSR-Net and SMART are 58-171 times the baseline, which again is to be expected due to the quadratic increase in the number of SSRs that have to be considered.

3.6.5 Energy Comparisons

In this section, we compare SHARP to SMART and SSR-Net in terms of energy consumption. In particular, we break down the average dynamic energy consumed per flit. For this analysis, we again assume an 8×8 network with an $HPC_{max} = 8$ for SMART and SSR-Net and an $HPC_{max} = 6$ for SHARP, which are the best achievable values for the proposed system configuration. We use the Uniform traffic pattern to determine the dynamic energy consumption. The results are shown in Fig. 3.23 for



(a) Wiring cost comparisons.



(b) SSR control logic area comparisons.

Figure 3.22: SSR arbitration wiring and logic area costs.

three injection rates, low, moderate, and high, which are set to 0.04, 0.14, and 0.24 flit/node/cycle, respectively. Energy results are provided for both the *prio_local* and *prio_bypass* schemes. The high injection rate is chosen based on the bypass saturation point for SSR-Net, which saturates the earliest. The results for each injection rate are normalized to the energy consumption of the SMART-8-local configuration at that injection rate. That is, we are using the SMART-8-local configuration as the baseline for comparison.

When the injection rate is low, we observe that SSR-Net and SHARP consume 7% less energy than SMART because of the energy reduction in the SSR arbitration logic. Moreover, the energy consumption is almost the same for both the *prio_local* and *prio_bypass* schemes at this injection rate.

When the injection rate is moderate, the contribution from the arbitration logic to the total energy consumption increases. For the *prio_local* scheme, the energy per flit reduces by 4% and 8% for SSR-Net and SHARP, respectively. For the *prio_bypass* policy, we see less energy per flit in buffering because longer SMART-hop paths are being established on average, which leads to a 16%, 21%, and 17% reduction in energy per flit for SMART, SSR-Net, and SHARP, respectively. We can infer from these results that for a moderate injection rate, the increase in energy consumption due to buffering is almost offset by the reduction in energy consumption for the SSR arbitration logic.

When the injection rate is high, the networks with the *prio_bypass* policy are near or at the saturation region, but the networks with the *prio_local* policy are still only moderately utilized. At this injection rate, SMART and SSR-Net both suffer from high rates of false negatives, which means that the SSR arbitration logic at most nodes

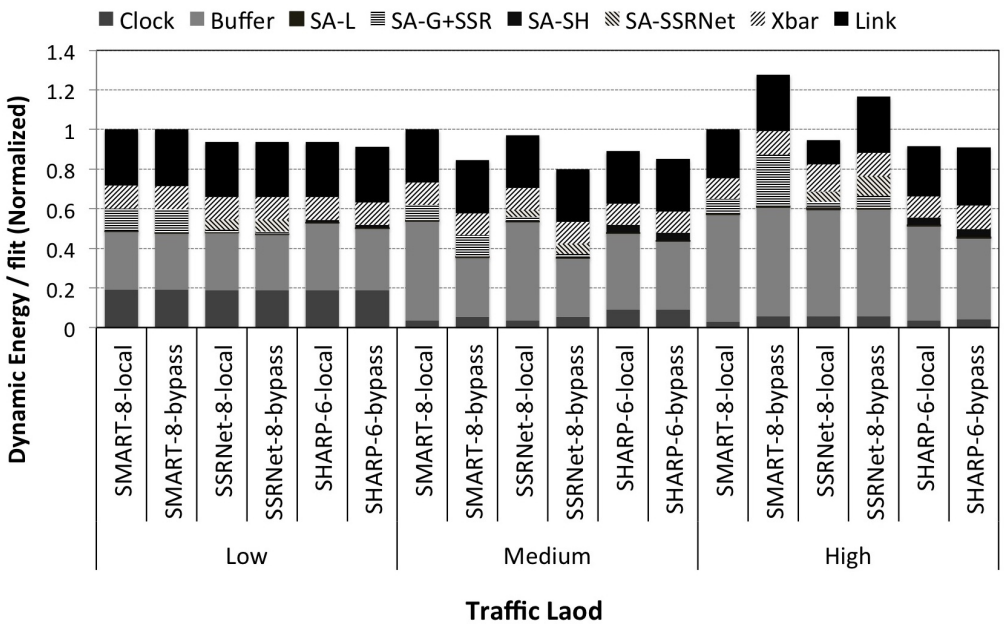


Figure 3.23: Energy consumption comparisons.

are active and consume power. Therefore, the arbitration logic has a larger share of the total dynamic energy per flit. Since the networks based on the *prio_bypass* policy suffer from a much higher rate of false negatives, their arbitration logic consumes 84-362% more energy than SMART-8-local. For SHARP configurations, a negligible part of the total energy comes from the arbitration logic since SHARP guarantees the avoidance of false negatives, and therefore it has a significantly simpler arbiter. Specifically, SHARP configurations reduce the dynamic energy by 7-9%, while SMART and SSR-Net with bypass policy consume 17-20% more energy, when compared to SMART-8-local. To summarize, the arbitration logic in SMART and SSR-Net consumes more energy, but they become increasingly inefficient at utilizing links effectively at higher injection rates.

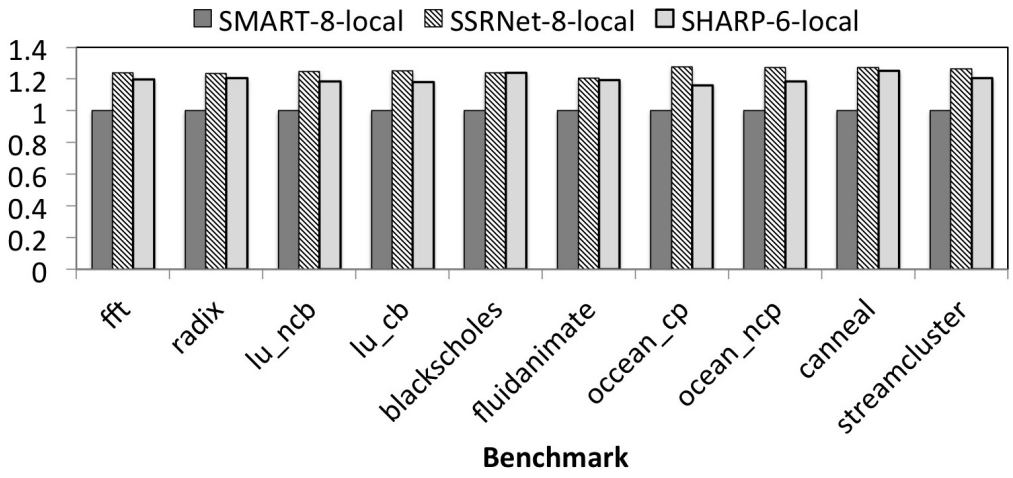


Figure 3.24: Full-system evaluation of real application benchmarks.

3.6.6 Full-System Evaluation

For full-system evaluation, we evaluate our design by running selected applications from the PARSEC and SPLASH-2 benchmark suites. We evaluate these benchmarks on a 64-core (8×8) system with 64 threads enabled, again using $HPC_{max} = 8$ for SMART and SSR-Net and an $HPC_{max} = 6$ for SHARP. We use the *prio_local* policy for these experiments. Fig. 3.24 shows the average flit latency comparisons for the three approaches. All results shown in Fig. 3.24 are normalized to the SMART-8-local configuration as the baseline.

For all benchmarks evaluated under full-system simulation, the injection rates fall in the lower end of the injection rates in the graphs for synthetic traffic. At these low injection rates, the higher HPC_{max} that SMART can achieve is beneficial in achieving a lower average flit latency since false negatives are less likely in this regime. This is why SHARP has a higher average flit latency than SMART. However, as observed in Section 3.6.2, the benefit of a higher HPC_{max} for SMART and SSR-Net quickly

diminishes with increasing injection rates due to the throughput loss caused by increasing probabilities of false negatives. The reason why SSR-Net has a higher average flit latency than SMART, despite having the same $HPC_{max} = 8$, is due to the extra pipeline stage in the SSR-Net design. For the *prio_bypass* policy, the full-system simulation results (not shown) are similar at these low injection rates for all three designs. Therefore, the relative comparisons are similar to the results shown in Fig. 3.24.

3.7 Chapter Summary

In this chapter, we presented an NoC design that uses asynchronous repeated wires to achieve single-cycle multi-hop traversals. In particular, we presented a design called SHARP that avoids several inherent shortcomings of an earlier design called SMART [45] that suffers from quadratic complexity and throughput loss due to false negatives. SHARP also ensures correctness when different priority modes are together in the same network.

In the next chapter, we present NoC designs for on-chip global communications that are based on the use of narrow-pitch repeated equalized transmission lines. These designs can transmit data across chip at extremely high data rates and low latencies. Also, these designs naturally support multicast and broadcast operations.

Chapter 3, in part, is in part, is currently being prepared for submission for publication of the material. Asgarieh, Yashar; Lin, Bill. The dissertation author was the primary investigator and author of this material.

Chapter 4

Transmission Lines-based NoC Designs

4.1 Introduction

This chapter explores NoC architectures based on the use of on-chip transmission lines (TLs) as a global shared medium [40, 41, 38, 77, 78, 20, 19, 68, 73]. Transmission lines can deliver data at the speed of light across the shared medium and consumes much less power than conventional wires because the wave propagation eliminates full-swing charges and discharges on the wire and gate capacitance. Transmission lines are attractive because they can provide very low latency packet delivery across chip (order of ns), very high bandwidths (20+ Gb/s per TL pair), and high energy efficiency.

Previously, Carpenter et al. [20, 19] proposed a globally shared-medium design for on-chip communications based on transmission lines. Their design comes with a number of limitations. First, they use transmission lines as a shared bus with differential

signaling, but their design does not make use of equalization circuitry or repeaters. To overcome frequency-dependent loss of transmission lines, their design requires wide-pitch transmission lines to ensure signal integrity at high data rates. The total pitch (including spacing and shielding) per differential pair of transmission lines is 45μm, which occupies considerable area[1].

Second, their bus-based architecture is merely a shared medium that allows configurable point-to-point communications, but does not support multicast or broadcast operations, which are critical for cache coherency protocol implementations. The reason for only allowing a single receiver is to limit the distortion that would be caused by multiple receivers when no equalization circuitry or repeaters are used[2].

Third, for access arbitration, the authors had assumed that a distributed arbitration protocol, such as carrier-sensing, would not be practical because some of the known protocols have poor bandwidth utilization properties. They cited a well-known collision detection protocol [59] as an example that can only achieve at most 36% bandwidth utilization. Therefore, they proposed instead to use a centralized scheduler for access arbitration. However, a key challenge in implementing a practical centralized arbitration scheme is the need for getting the *requests* from the cores to the centralized scheduler and the *grants* back to the cores. Unfortunately, if significant latencies are incurred on these control lines, the performance of a centralized scheduler diminishes substantially, lead-

[1] As will be discussed in Section 4.2, the total pitch for our design (including spacing and shielding) is only 7.8μm, which is about 6x narrower.

[2] Previously, Ito et al. [38] proposed a bidirectional multi-drop transmission line interconnect that can support multiple simultaneous receivers, but these links can only reliably operate at much lower data rates (e.g., 8 Gb/s) due to attenuation caused by multiple receivers when no equalization circuitry or repeater structures are used. Further, their design is also based on wide-pitch transmission lines that have similarly significant area overhead. Carpenter et al. [19] suggested that their design may be able to support two receivers with tolerable distortion, but their design still does not support multicast or broadcast in general.

ing to substantially diminished system performance. In addition, their design requires a separate receiver wake-up operation, which is again coordinated by the centralized scheduler. Potentially significant latencies may also be incurred on the corresponding control lines, which could lead to further degradation in system performance.

In this chapter, we propose several novel designs for global on-chip communications based on repeated equalized transmission lines (RETLs) that overcome the above limitations. Our contributions can be summarized as follows:

- We propose designs based on shorter-length segments of transmission lines that are *repeated* and *equalized* at each segment to form a global interconnect. The use of sophisticated equalization circuitry and a repeater structure [77, 78, 68] enables the use of *thin* wires to implement the transmission lines that can tolerate the resistive loss and inter-symbol interference at very high data rates (e.g., 20 Gb/s per TL pair with differential signaling).

- In particular, we propose several designs based on the use of RETLs to form a shared global bus. In these designs, we allow all receivers to *simultaneously listen* to the shared medium while ensuring signal integrity, which means multicast and broadcast operations can readily be supported. These operations are essential for cache coherency protocol implementations.

- For this shared RETL bus approach, we propose several novel arbitration schemes that are *fully distributed*. These distributed schemes can achieve very high throughput and bandwidth utilization. In particular, we propose a token-based arbitration scheme that is well-suited to TLs and a distributed randomized polling scheme,

both of which are simple to implement. Unlike arbitration schemes that have been proposed for nanophotonics that rely on the inherent ability of nanophotonics to *divert* light [71, 70], for which there is no equivalent for transmission lines, the schemes that we propose are based on the ability of multiple receivers to determine when a channel becomes available and ensure that only one sender attempts to transmit at a time.

- We further show how the performance of the distributed token-based and randomized polling arbitration schemes can be improved by means of spatial partitioning.

- Besides the shared RETL bus approach, we also describe a dedicated interconnection architecture that supports multicast and broadcast operations as well. This design does not require arbitration.

- For designs based on either the shared bus or the dedicated interconnection approach, we demonstrate solutions that can each provide 640 Gb/s aggregated throughput and enable communications between any on-chip cores in only one or two core clock cycles. Given the narrow pitch of RETLs, our design can easily scale to multiple terabits per seconds with additional lanes.

- Simulation results with both synthetic and real benchmarks with up to 64 parallel threads demonstrate that our proposed solutions are capable of achieving high performance.

The rest of the chapter is organized as follows: Section 4.2 presents an overview of the repeated equalized transmission lines (RETLs) that we use in our designs. Section 4.3 presents an overview of our proposed global shared medium architecture and

how it operates. Section 4.4 presents several novel fully distributed arbitration schemes for coordinating access to this shared RETL medium. Section 4.5 presents an alternative approach based on a dedicated interconnection architecture. Section 4.6 presents evaluation results, and Section 4.7 concludes the chapter.

4.2 Repeated Equalized Transmission Lines

Unlike the wide-pitch TLs used in previous works (e.g., [38, 20, 19]), we utilize equalization techniques in our TL structure in order to significantly reduce the pitch of the TLs as well as to achieve a high reliable data rate. In particular, we utilize the repeated equalized transmission line (RETL) design from our previous work [77, 78, 68]. We defer the reader to Appendix B for a more detailed discussion of this design. Here, we simply depict an abstraction of a point-to-point RETL segment in Fig. 4.1. Unlike conventional wires, a transmission line operates in the LC region at high-frequency (e.g. 20+ GHz) and carries low-swing waveforms at near the speed of light. The RETL design employed in this paper is based on a differential TL pair with terminated resistance, with the TL pair surrounded by power and ground lines for shielding. As described in details in Appendix B, the transmitter (Tx) component depicted in Fig. 4.1 corresponds to the chain of tapered current-mode logic (CML) buffers shown in Fig. B.2 in Appendix B. This chain of tapered CML buffers acts the driver. The number of CML stages and the tapered factor can be optimized based on the length of the TL segment and the expected load. On the receiver side, the receiver (Rx) component depicted in Fig. 4.1 corresponds to the continuous-time linear equalizer (CTLE) and the sense-amplifier based latch shown in Fig. B.2 in Appendix B. The CTLE and the sense-amplifier based latch are used

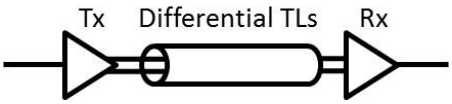


Figure 4.1: Abstraction of a point-to-point RETL segment.

to recover the transmitted signal by boosting the eye-opening (i.e., to compensate for resistive loss and inter-symbol interference).

By co-optimizing the transmitter, the length and pitch of the differential TL pair, and the receiver together, we can achieve an optimized result for a target link throughput and latency. The reader is referred to Appendix B for a detailed description of this co-optimization flow. With this co-optimization flow, we can derive an RETL segment that is 2.5mm in length, which can achieve a 20 Gb/s throughput and a 40 ps/mm normalized latency. Equivalently, an RETL segment operates at a 20 GHz communication clock frequency. The performance metrics for this link segment design is shown in Table 4.1. With the use of equalization techniques, *thin* wires can be used to implement the TLs. Our design supports a wire pitch of 2.6μm that includes the wire width and the wire spacing. The total pitch is 7.8μm, which includes the wires and spacing for a pair of differential TLs as well as the surrounding power and ground lines for shielding. This is about 6x narrower than the total pitch of 45μm required for the wide-pitch TLs used in previous works (e.g., [20, 19]). Note that these RETL segments are *unidirectional*. They can be cascaded together to form longer connections.

Table 4.1: RETL performance metrics.

Dimensions	Pitch (width+spacing): 2.6μm, Total pitch(a pair of differential TLs + power/ground shielding): 7.8μm, Length: 2.5mm each segment.
Delay	Transmission line: 13 ps, Tx: 44 ps, Rx: 45 ps, Total: 102 ps, Normalized: 40 ps/mm.
Power	Total: 1.66 mW (Tx: 0.79 mW, Rx: 0.87 mW), Energy/bit: 0.08 pJ/b.

4.3 Shared Medium Architecture

In this section, we describe how the RETLs described in the previous section can be used to build a global shared RETL medium. In particular, we describe how the overall system is organized with respect to the processor cores and this shared medium. We also describe in this section how data transmission over this shared RETL medium operates.

4.3.1 Cluster Architecture

In this section, we describe the design of a system that comprises 64 cores. Each core comprises a processor, an L1 data/instruction cache, a slice of a shared L2 cache, and a slice of the cache coherency directory. To take full advantage of the high data rates that can be achieved over the shared global RETL medium, we group four cores together into clusters via 4:1 concentrators. This is depicted in Fig. 4.2. Each cluster of four cores shares a common interface to the shared global RETL medium for inter-cluster communications. All inter-cluster traffic will be transmitted via the shared global RETL medium in FIFO order. For *intra-cluster* traffic, they will be handled by the concentrator,

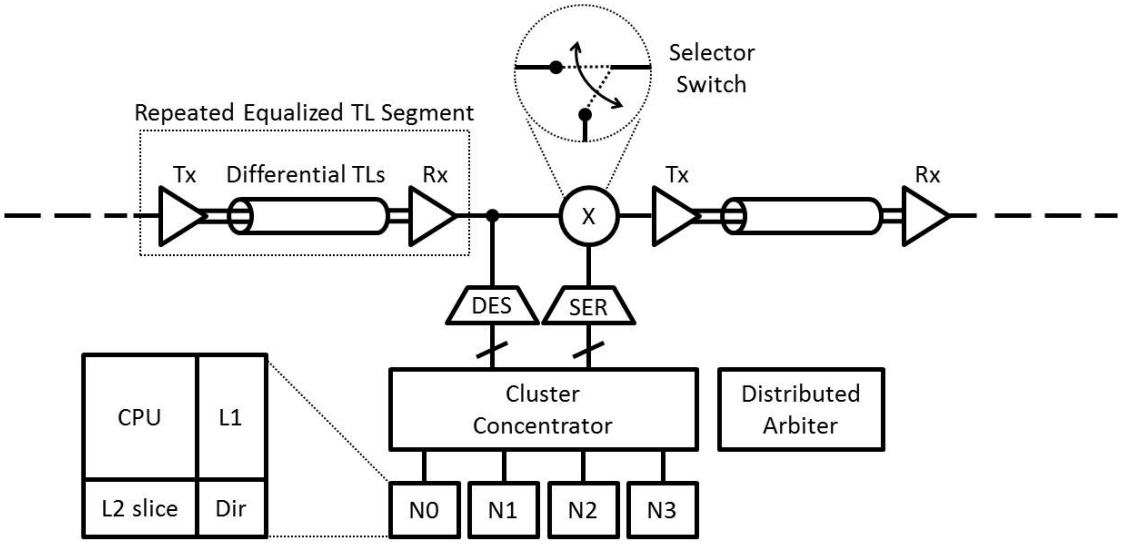


Figure 4.2: Clusters of four processor cores are grouped together via 4:1 concentrators. Clusters are interconnected by RETL segments.

which also acts as a local crossbar. For cache coherence, we assume a MOESI directory-based protocol over a snoopy-based approach because a MOESI directory-based approach has been shown to be more powerful and performance efficient [32].

For our evaluations in Section 4.6, we assume a core clock frequency of 1.25 GHz, which corresponds to a 0.8 ns clock period. With the shared RETL medium operating at a 20 GHz communication clock frequency, data can be sent 16 times faster over the shared RETL medium than the core clock rate. To bridge the two clock rates, 16:1 serializers (SER) and de-serializers (DES) are shown in Fig. 4.2 to perform the respective operations. These serializers/de-serializers can be efficiently implemented using a power-efficient tree structure that includes a four-level tree of buffers and dividers [42].

The clusters are interconnected via cascaded RETL segments that form a global shared medium. When a cluster transmits data on this shared medium, all clusters can simultaneously *listen* to this shared medium, and each cluster can determine for

itself if it should receive the data being transmitted based on the addressing information provided. Since all clusters can simultaneously listen to and receive from the shared medium, multicast and broadcast operations can readily be supported.

To ensure that only one cluster can transmit at a time, an access arbitration mechanism is needed. Our approach is based on fully *distributed* arbitration schemes. Each cluster employs a *local* arbiter that independently determines when the cluster can transmit. We defer to Section 4.4 for descriptions of several novel distributed arbitration schemes for coordinating access to the shared medium.

4.3.2 Shared RETL Bus

For a system with 64 cores, they are grouped together into 16 clusters. Fig. 4.3(a) shows how the 16 clusters are interconnected together to form a unidirectional *ring*. Fig. 4.3(b) depicts the ring topology in a chip layout with the 16 clusters. At any moment in time, the loop is broken by the cluster that is transmitting to form a unidirectional *bus*. The unidirectional bus starts at the transmitting cluster i and ends at cluster $((i+N-1) \bmod N)$, where N is the number of clusters. This is achievable because two cascaded RETL segments are separated by a *selector switch*. The transmitting cluster breaks the loop by setting its selector switch. This selector switch is depicted in Fig. 4.2. Just before a cluster starts transmitting, it configures its selector switch to connect the output of the serializer (SER) to the RETL transmitter (Tx). This selector switch configuration also *disconnects* the preceding RETL segment from the forwarding path, hence breaking the loop. All other non-transmitting clusters have their selector switches configured for pass-through. The selector switch can be implemented using a standard CMOS pass-gate

structure, which can be quickly reconfigured for connection and disconnection.

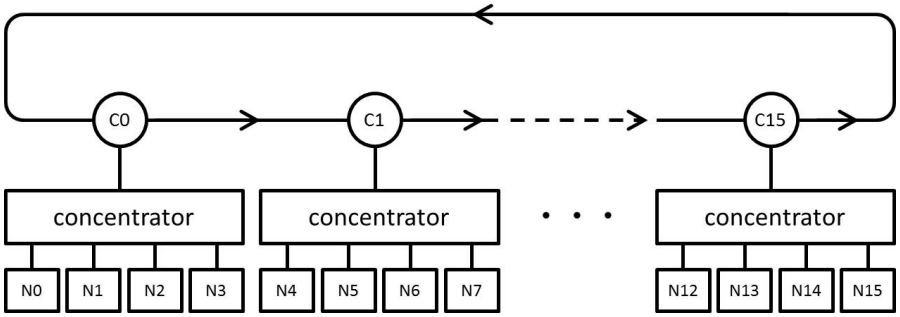
We assume a chip size of 10mm×10mm, divided into sixteen 2.5mm×2.5mm clusters, with each of the four cores in a cluster occupying a 1.25mm×1.25mm area. This means the sixteen clusters can be interconnected by cascading sixteen 2.5mm RETL segments to form a ring. However, with the decoupling of the ring by transmitting cluster i, the longest distance that a signal needs to travel to reach the last reachable cluster $((i + N - 1) \mod N)$ is only $N - 1 = 15$ RETL segments away, or 37.5mm. With a normalized latency of 40 ps/mm, any cluster can reach any other cluster in just 1.5 ns, or under two core clock cycles at 1.25 GHz[3].

For our evaluations in Section 4.6, we assume a design with 32 lanes of transmission lines, each lane capable of sending 20 Gb/s, which provides an aggregated throughput of 640 Gb/s. This means 64 bytes (which corresponds nicely to a cache line) may be sent over the shared RETL bus per core clock cycle. This design provides ample bandwidth for the benchmarks evaluated. Given the narrow pitch of our RETLs, our design can easily scale to multiple terabits per seconds with additional lanes. For example, a design with 128 lanes can achieve an aggregated throughput of 2.56 Tb/s.

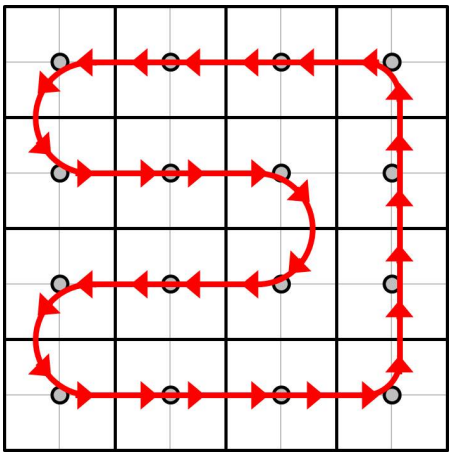
4.3.3 Timing of Operations

We now examine the timing of data transmissions over the proposed shared RETL bus. The timing of data transmission operations is depicted in Fig. 4.4. As will be discussed next in Section 4.4, each cluster will monitor the shared RETL bus to determine whether or not it should start transmitting in the next core clock cycle. If it determines that it should start transmitting in the next core clock cycle, it configures

[3]Two core clock cycles at 1.25 GHz is 1.6 ns.



(a) Topological abstraction showing how the clusters are interconnected.



(b) Chip layout organization for a 64-core system organized into 16 clusters.

Figure 4.3: Overall system organization. The shared global RETL bus is unidirectional. Though the diagrams show the shared global RETL medium forming a ring, the loop is broken by the transmitting cluster via setting its selector switch accordingly.

its selector switch to disconnect the connection from the preceding RETL segment and instead redirects the connection from the output of its serializer (SER) to the RETL transmitter (Tx) of the next RETL segment. This selector switch configuration occurs shortly before the start of the next core clock cycle. This way, at the start of the next core clock cycle, it can start transmitting data on the shared RETL bus without a loop.

As shown in Fig. 4.4, cluster i transmits data to cluster a in the next core clock cycle, followed by data to cluster b in the following cycle, and data to cluster c in the cycle after. Cluster i can continue to send data to different receivers (or possibly multicast or broadcast to multiple receivers) as long as it has possession of the shared RETL bus. When cluster i finishes transmitting data, the shared RETL bus will need to be *idle* for a period of time to allow the signals to *drain* through the shared medium. Recall that a transmission line works by transmitting low-swing waveforms at very high frequencies (e.g., 20 GHz). These waves propagate through the shared transmission line medium. The next cluster cannot safely start transmitting on this shared transmission line medium until all *in-flight* waves have propagated through.

Recall from Section 4.3.2 that the longest distance that a wave needs to propagate is through 15 RETL segments, or a worst-case distance of 37.5mm, which takes 1.5 ns. Therefore, the worst-case draining period is under a two-cycle turnaround time. Referring again to Fig. 4.4, cluster j can start transmitting two cycles after cluster i stops transmitting. As the worst-case draining period is under two core clock cycles, cluster j can configure its selector switch shortly before the end of the two cycles so that it can start transmitting at the start of the next clock cycle, which is to cluster d, followed by cluster e.

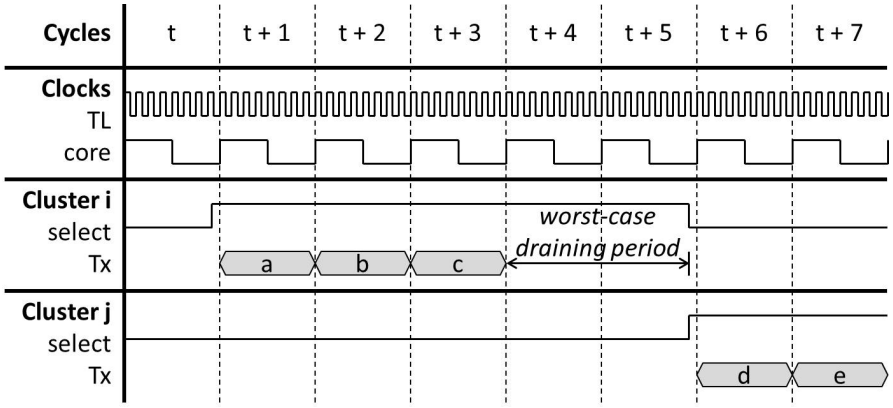


Figure 4.4: Timing of operations for the proposed shared RETL bus.

4.4 Distributed Arbitration for RETL Bus Sharing

In this section, we provide several fully distributed schemes for solving the arbitration problem needed for our shared RETL bus[4] approach. Our shared RETL bus approach requires arbitration to prevent two or more clusters from transmitting at the same time. We first describe a token-based arbitration scheme similar to what has been used in token ring LAN systems [9], but our version of the scheme has been designed to fit well with the characteristics of our shared RETL bus. We then describe another scheme based on the idea of *distributed randomized polling*. In addition, we extend both schemes to work with multiple shared RETL buses that operate in parallel. In particular, multiple buses could be implemented using the same number of lanes by spatially partitioning the lanes into multiple buses so that each (narrower) bus is implemented with fewer lanes. The use of multiple parallel buses enables multiple clusters to transmit concurrently. As we shall see in Section 4.6, better performance can be achieved with multiple parallel buses even when the number of lanes used remains the same.

[4]Throughout this section, we will occasionally refer to a shared RETL bus simply as a bus.

4.4.1 Token-Based Arbitration

A typical token ring scheme [9] is based on an acquire-and-release mechanism. A token gets circulated from one sender (cluster in our case) to the next until it is *acquired* by a sender that has data to send (i.e., it has a non-empty queue). Such a sender is called a *requester*. The sender then transmits the data that it wishes to send, possibly for multiple clock cycles, depending on how much data that it wants to send. When it finishes sending the data, it *releases* the token for circulation to other senders. The token bypasses *non-requesters* (i.e., senders with empty queues) until it is acquired by a sender that has data to send.

A straightforward implementation in our setting would be to implement the control token ring as a one-bit ring with conventional wires and latches, where a one-bit token would circulate from one cluster to the next at each clock cycle until the token is acquired by a requesting cluster. Once a requesting cluster has acquired the token, it may take multiple cycles to transmit its data, followed by a two-cycle draining period. After which, it releases the token, which may take multiple cycles to circulate through the control token ring until the token reaches the next requesting cluster. A problem with this approach is that it may take multiple cycles before the token reaches the next requesting cluster, which would leave the bus unnecessarily idle.

Alternatively, token-based arbitration schemes have been proposed for nanophotonics [71, 70]. In these schemes, the token is broadcasted on an optical ring. These schemes rely on the inherent ability of nanophotonics to *divert* light. That is, the token travels along the optical ring, bypassing non-requester, until it is diverted by a requester, at which point the light is completely removed from the optical ring to provide an exclu-

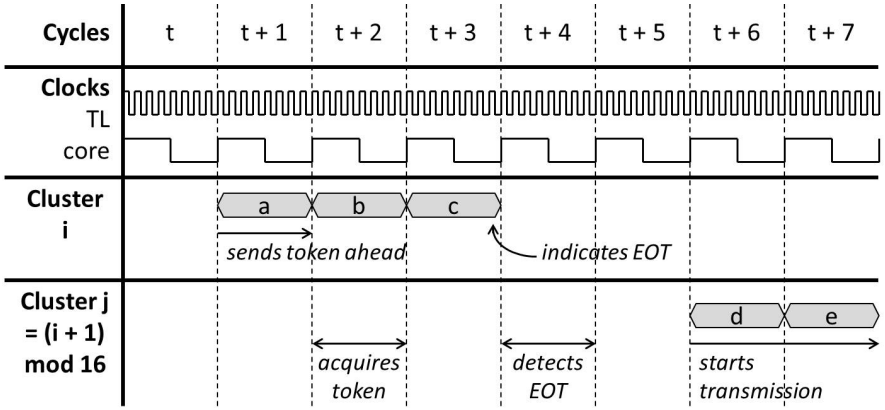


Figure 4.5: Example of token-based arbitration timing.

sive grant for the corresponding optical channel. Unfortunately, there is no equivalent diverting ability for transmission lines.

In our approach, we also implement the control token ring as a one-bit ring with conventional wires and latches. However, we extend this straightforward scheme with three ideas:

- *Send-ahead tokens*: We allow the current cluster which has acquired the bus to send data for up to T consecutive cycles. In our design with 32 lanes at 20 Gb/s (640 Gb/s aggregated throughput), 64-bytes can be transmitted per core clock cycle at 1.25 GHz. Consider the example depicted in Fig. 4.5. At cycle $t+1$, cluster i has already acquired the bus and transmits to cluster a. At the same cycle, cluster i *sends ahead* the token along the control token ring. Suppose cluster j, where $j = i+1$, is a requesting cluster. Then it receives and acquires the token one cycle later at $t+2$. Meanwhile, cluster i continues to transmit data to cluster b in cycle $t+2$ and cluster c in cycle $t+3$.

- *Completion sensing and explicit EOT indication*: The second idea is *completion*

sensing. Although cluster j has acquired the token in cycle $t_{acquire} = t + 2$, it does not start transmitting. Instead, it *monitors* the bus to determine when the previous transmitter has finished. Rather than inferring completion by monitoring for idle cycles, we require cluster i to explicitly send an "End-of-Transmission" (EOT) status bit at the end of the cycle to indicate that it has completed its transmission. Referring again to Fig. 4.5, the EOT status bit is sent at the end of cycle $t + 3$.

- *Relative propagation time*: The third idea is to help cluster j decide when it can start transmitting after it has detected the EOT status bit. As discussed in Section 4.3.2, in our design, the longest distance that a signal needs to travel to reach the last reachable cluster is at most $N - 1 = 15$ segments away, corresponding to the propagation delay through 15 RETL segments, or 37.5mm, which takes just under two core clock cycles. Since the EOT status bit is sent along the transmission lines, it can take up to almost two core clock cycles for a cluster to detect the EOT status bit. However, clusters that are closer to the last transmitting cluster may detect the EOT status bit after just one core clock cycle. In particular, for clusters that are less than $N/2$ segments away, they will detect the EOT status bit just after one cycle, whereas clusters that are greater or equal to $N/2$ segments away will detect the EOT status bit two cycles later.

  In the example depicted in Fig. 4.5, since cluster j is only one segment away, it will detect the EOT bit in cycle $t_{detect} = t + 4$, one cycle after cycle $t + 3$ when cluster i sent the EOT bit. Since cluster j has been monitoring the bus, it also knows that the last transmitting cluster is cluster i since the data transmitted contains

both the source ID and the destination ID. Based on a table lookup, cluster j can determine if it should check if it has already acquired the token already in cycle $t_{check} = t_{detect}$, or if it should wait for another cycle until $t_{check} = t_{detect} + 1$ for the data sent by cluster i to drain through the system before checking if it has acquired the token. In this example, cluster j knows that it is less than $N/2$ segments away from cluster i, and therefore, it will wait another cycle to $t_{check} = t + 5$ to check if it has the token, and it will start transmitting at the next cycle $t + 6$. In another words, depending on the relative position of cluster j to cluster i, t_{check} is either t_{detect} or $t_{detect} + 1$. As depicted in Fig. 4.5, cluster j starts transmitting to cluster d at cycle $t + 6$, then to cluster e in cycle $t + 7$, and so on.

It is worth noting that with the explicit EOT indication, a cluster that has already acquired the token can start transmitting at most two cycles after the last transmitting cluster has finished. Without the explicit EOT indication, a cluster would have to wait for one more cycle to be sure that the last transmitting cluster had finished – two cycles to ensure all in-flight waves have drain through the system, plus another cycle to make sure that bus has been idled.

If the token has not reached a requesting cluster (a cluster with a non-empty queue) by the end of cycle t_{check}. then all requesting clusters will monitor the bus for two cycles to check if the bus is idle. If one of the requesting clusters has acquired the token during these interim two cycles, then it will start transmission after these two cycles. Otherwise, all requesting clusters will repeat monitoring for two more cycles until one of the requesting clusters has acquired the token.

In our evaluation in Section 4.6, we assume the control token ring is clocked with

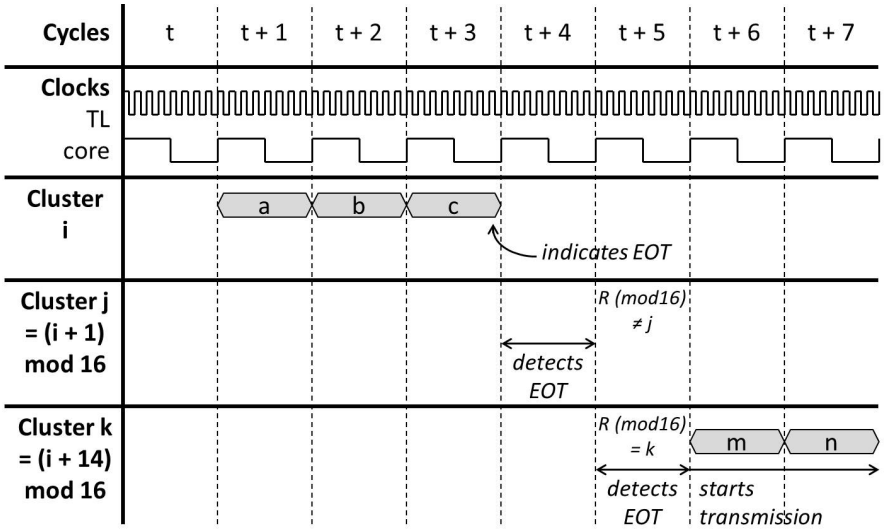


Figure 4.6: Example of distributed randomized polling-based arbitration timing.

the core clock. Conceivably a faster clock could be used, in which case, the performance may improve.

4.4.2 Distributed Randomized Polling

Instead of using a token ring, we can alternatively employ the concept of *distributed randomized polling*, based on the following ideas:

- Like the token-based arbitration scheme described above, we also rely here on completion sensing and explicit EOT indication. Consider the example depicted in Fig. 4.6. Suppose the current cluster that has acquired the bus is again cluster i, and it transmits to cluster a in cycle $t + 1$, cluster b in cycle $t + 2$, and cluster c in cycle $t + 3$.

- At the end of the cycle $t + 3$, we again require cluster i to send an EOT status bit. Meanwhile, all other clusters monitor the bus to detect the EOT status bit.

Again, depending on the relative distance that a cluster is to the last transmitting cluster. In the example depicted in Fig. 4.6, cluster j is one segment away, so it will detect the EOT status one cycle later in cycle $t_{detect} = t + 4$. On the other hand, cluster k is 14 segments away, so it will detect the EOT status two cycles later in cycle $t_{detect} = t + 5$.

- Rather than using a token passing mechanism and having the clusters check if they have acquired the token, we use a pseudo-random number generator to *poll* the clusters. In particular, all clusters will implement the same pseudo-random number generator logic, for example using a *linear feedback shift-register* [29], which can be implemented with negligible cost. A pseudo-random number generator will generate a random sequence of numbers, changing from one random number to another random number each clock cycle. To ensure that all clusters will see exactly the same random number sequence, all the pseudo-random number generators can be initialized to the same seed.

- With the availability of a pseudo-random number generator at each cluster, each cluster will see the same random number R in each cycle. If a cluster is less than $N/2$ segments away from the last transmitting cluster, for example cluster i in Fig. 4.6, then it will *poll* the random number R in cycle $t_{check} = t_{detect} + 1$. On the other hand, if a cluster is greater or equal to $N/2$ segments away, for example cluster k in Fig. 4.6, then it will poll the random number R in cycle $t_{check} = t_{detect}$. Each cluster can detect how far it is away from the last transmitter by doing a table lookup since it knows the source ID of the last transmitter.

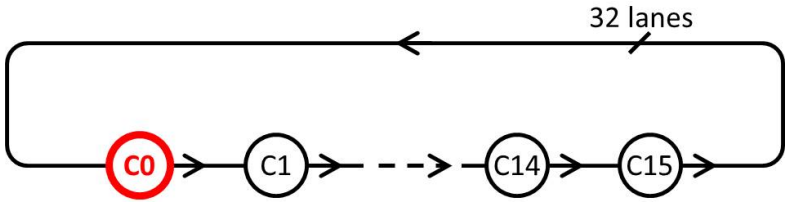
- In Fig. 4.6, both cluster j and cluster k will poll R in cycle $t + 5$ to see if (R mod N) is equal to its cluster ID. In this example, cluster k matches (R mod N), with $N = 16$. Therefore, it starts transmission in the next cycle $t + 6$ to cluster m, then to cluster n in cycle $t + 7$, and so on.

Like the token-based arbitration scheme, it is possible that the cluster that matches the random number R at cycle t_{check} has an empty queue. In this case, all clusters will monitor the bus for two cycles to check if the bus remains idle. If the bus has remained idle, then again all clusters will check their ID against the random number of R. If the matching cluster is still empty, then all clusters will repeat monitoring for two more cycles until one of the clusters that matches R is non-empty.

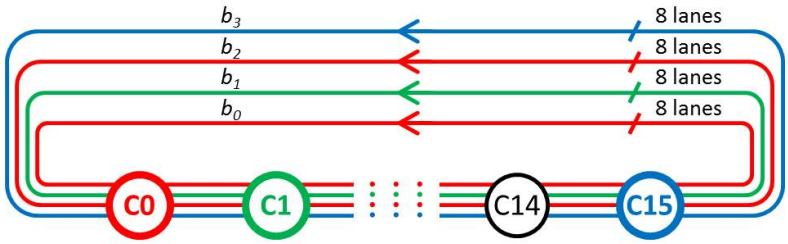
4.4.3 Spatial Partitioning

In this section, we partition the bus into multiple narrower buses that operate in parallel, and we extend both arbitration schemes to work in parallel buses. By having multiple parallel buses, multiple clusters can transmit concurrently. Consider the example depicted in Fig. 4.7. In our evaluation, we assume a 32-lane shared RETL bus, where each lane provides 20 Gb/s of data rate, for an aggregated throughput of 640 Gb/s. At the core clock frequency of 1.25 GHz, the 32-lane shared bus can transmit 64 bytes per cycle. This configuration is depicted in Fig. 4.7(a), which shows cluster 0 transmitting.

Fig. 4.7(b) depicts a spatial partitioning of the 32 lanes into four parallel (but narrower) 8-lane shared buses, b_0, b_1, b_2, and b_3. Instead of transmitting 64 bytes per core clock cycle, only 16 bytes can be transmitted over one of the four buses per core clock cycle. Though each of the four parallel buses provides less capacity, they permit



(a) Single shared RETL bus.



(b) Multiple parallel (but narrower) RETL buses.

Figure 4.7: With spatial partitioning into multiple parallel (but narrower) RETL buses. Multiple clusters can simultaneously transmit. (a) Cluster 0 transmits over all 32 lanes. (b) Cluster 0 transmits over two 8-lane buses b_0 and b_2 (shown in red), cluster 1 transmits over one 8-lane bus b_1 (shown in green), and cluster 15 transmits over one 8-lane bus b_3 (shown in blue).

up to four clusters to transmit concurrently. If the traffic load is low-to-moderate, then we want an arbitration mechanism that will allow one cluster to use more than one bus (possibly all four buses) concurrently. Fig. 4.7(b) depicts cluster 0 transmitting over two 8-lane buses b_0 and b_2 (shown in red), cluster 1 transmitting over one 8-lane bus b_1 (shown in green), and cluster 15 transmitting over one 8-lane bus b_3 (shown in blue). If a cluster has acquired more than one bus, then it will load-balance its traffic across the acquired buses. The 32 lanes can be partitioned into different number of parallel buses of different widths, for example, two 16-lane parallel buses or eight 4-lane parallel buses. Given the narrow pitch of RETLs, we can also add more parallel buses with more lanes per bus, as space permits. For our evaluations in Section 4.6, partitioning 32 lanes was found to be adequate.

Extending token-based arbitration

Using the example of four parallel buses shown in Fig. 4.7(b), we can extend the token-based arbitration scheme by implementing four separate control token rings, one per parallel bus. The operation and timing of each parallel bus would be the same as explained in Section 4.4.1. If a cluster has acquired more than one bus, then it will load-balance its traffic across the buses acquired. We can initialize the starting token position to a random location for the four control token rings.

One problem with this approach is the following: clusters closer to the currently active clusters have priority over clusters that farther downstream in acquiring tokens. If a cluster has acquired multiple buses, then the next requesting cluster downstream will likely acquire the same bundle of buses when the currently active cluster finishes.

This *synchronization* could make it increasingly unlikely that multiple clusters will be transmitting simultaneously over different buses. It is conceivable that all four buses would only be used by one cluster at a time. As we shall see in Section 4.6, this extension does not perform better than without spatial partitioning.

Extending distributed randomized polling

We can also similarly extend the distributed randomized polling-based arbitration scheme by implementing four separate pseudo-random number generators at each cluster, one corresponding to each of the parallel buses (e.g., R_0, R_1, R_2, and R_3 for buses b_0, b_1, b_2, and b_3, respectively). The operation and timing of each parallel bus would be the same as explained in Section 4.4.2. If a cluster has acquired more than one bus, then it will load-balance its traffic across the acquired buses as well.

Unlike the token-based arbitration scheme where clusters closer to the currently-active clusters have a higher priority, the randomized nature of the polling scheme means that all clusters have equal probability of acquiring each of the four buses, which makes it unlikely that any one cluster would acquire multiple buses when there are other clusters contending for them. As we shall see in Section 4.6, this spatial partitioning extension of the distributed randomized polling method leads to better results.

4.5 Dedicated Interconnection Architecture

In this section, we present an alternative design based on a dedicated interconnection architecture. As with our shared RETL bus designs described in Sections 4.3 and 4.4, we also base our design in this section on a system that comprises 64 cores that are

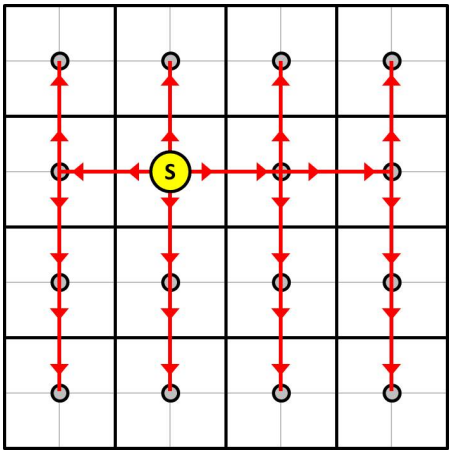


Figure 4.8: Each cluster has its own dedicated tree-based broadcast network.

organized into sixteen clusters across a 10mm×10mm chip, with each of the four cores in a cluster occupying a 2.5mm×2.5mm area, and each core occupying a 1.25mm×1.25mm area. Each core again comprises a processor, an L1 data/instruction cache, a slice of a shared L2 cache, and a slice of the cache coherence directory, all of which operate on a 1.25 GHz core clock. We also base on our design on RETL segments that operate on a 20 GHz communication clock.

However, in contrast to the shared RETL bus designs described in Sections 4.3 and 4.4, we describe in this section a design in which each cluster has its own dedicated *tree-based* broadcast network, on which *only* the associated cluster can transmit as the source. As such, there is no need for arbitration or wait for a draining period. To form a broadcast tree, RETL segments are cascaded together. All other clusters are connected to this tree and can act as receivers. This way, the source cluster can transmit to any cluster, multicast to multiple clusters, or broadcast to all clusters. Fig. 4.9 depicts how a cluster connects to its own dedicated tree-based broadcast network. Similar to our bus-based designs, intra-cluster traffic are handled by its concentrator, which acts as a local

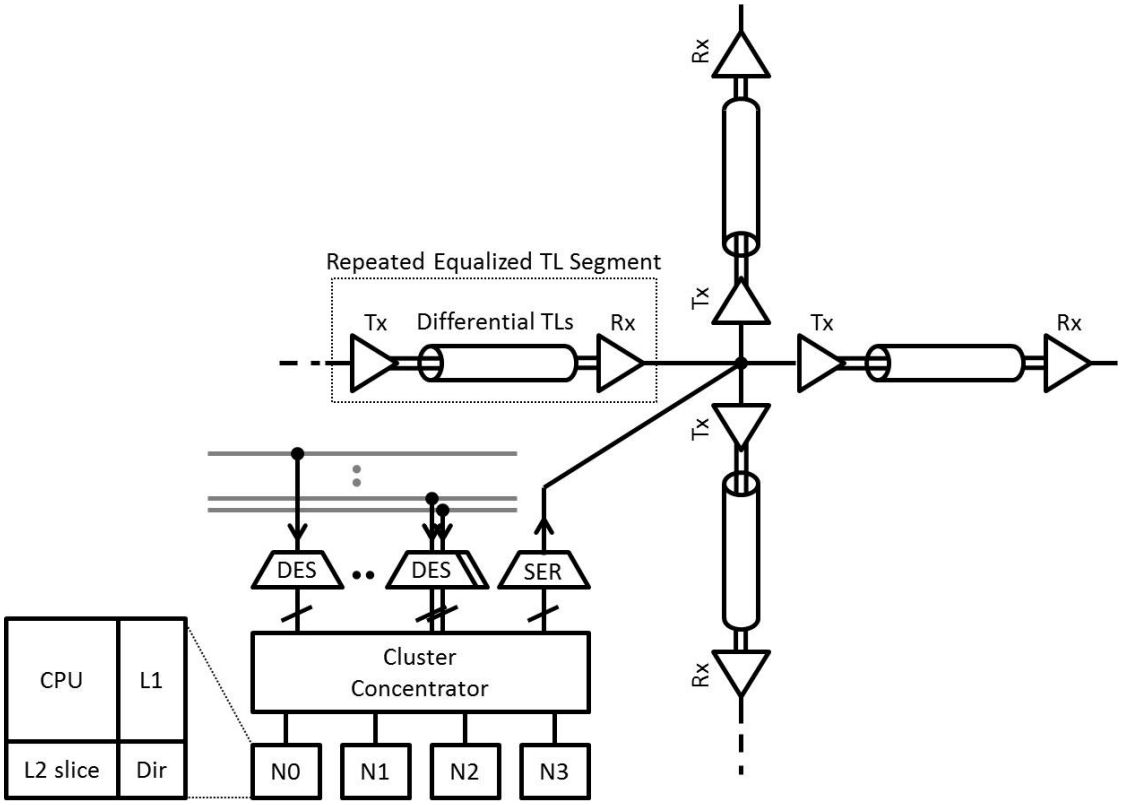


Figure 4.9: Architecture of a cluster and how it connects to its own dedicated tree-based broadcast network.

crossbar, and inter-cluster traffic are handled by the corresponding dedicated broadcast tree.

For a design with 16 clusters, we have 16 *separate* dedicated broadcast trees. Each cluster can receive from all other $N - 1 = 15$ clusters, as depicted in Fig. 4.9. For our evaluations in Section 4.6, we assume each dedicated broadcast tree uses 2 lanes of transmission lines, each lane capable of sending 20 Gb/s. This way, with 16 clusters, we utilize a total of $16 \times 2 = 32$ lanes of transmission lines so that this dedicated interconnection architecture design can be compared with our shared RETL bus designs, which also use 32 lanes of transmission lines. The 16 broadcast trees, with 2 lanes each,

together provides an aggregated throughput of 640 Gb/s, just like our shared RETL bus designs. Given the narrow pitch of our RETLs, our proposed dedicated interconnection architecture can easily scale to multiple terabits per seconds by adding more lanes to each dedicated broadcast tree.

Note that in our shared RETL bus designs, each lane of the bus comprises 16 RETL segments to form a ring. Therefore, 32 lanes require 32×16 = 512 RETL segments. In our dedicated interconnection architecture design, each lane of our dedicated broadcast network comprises 15 RETL segments to form a tree. Therefore, 16 dedicated broadcast trees with 2 lanes each require (16×2)×15 = 480 RETL segments, so both types of architectures employ a comparable number of RETL segments. For both types of architectures, 2.5mm RETL segments are needed to span the width or height of a cluster. We further note that with a broadcast tree structure, the longest distance that a signal needs to travel is 6 RETL segments, or 15mm (or more generally $2 \times (\sqrt{N} - 1)$ RETL segments). This means that the worst-case signal propagation latency is well under one core clock cycle.

With the layout shown in Fig. 4.8, each cluster has 16×2 = 32 lanes of transmission lines crossing at least one of its edges in the vertical direction. With a total pitch of 7.8μm per lane, 32 lanes×7.8μm = 0.25mm is well under the 2.5mm width of a cluster. An alternative layout is shown in Fig. 4.10. In this layout, half of the lanes for each broadcast tree is laid out in the *vertical* direction, as shown in Fig. 4.10(a), and the other half of the lanes is laid out in *horizontal* direction, as shown in Fig. 4.10(b). In a design with 2 lanes per broadcast tree, one of these lanes would be laid out in the vertical direction, and the other lane would be laid out in horizontal direction. This way,

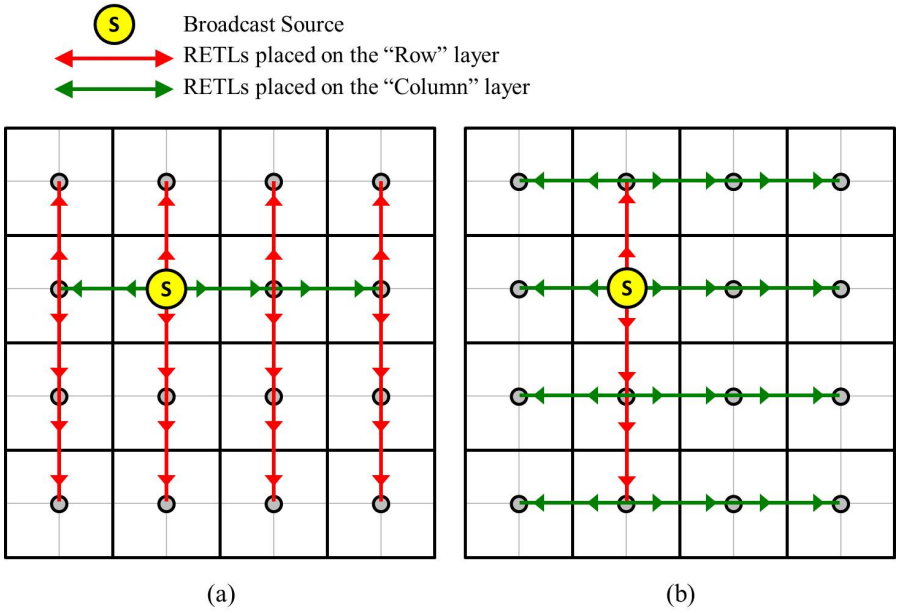


Figure 4.10: (a) Tree layout in the vertical direction. (b) Tree layout in the horizontal direction.

only $16 + \sqrt{16} = 20$ lanes (or more generally, $N + \sqrt{N}$ lanes) would cross each edge of a cluster. Again, given the narrow pitch of our RETLs, many more lanes could be added to provide higher throughput, and the orthogonal layout approach depicted in Fig. 4.10 enables more lanes per broadcast tree or a narrower cross-section for the same number of lanes.

4.6 Evaluation

4.6.1 Experimental Setup

Our experimental setup follows the baseline designs described in Sections 4.3, 4.4, and 4.5. In particular, we assume a design that comprises 64 cores that are organized into 16 clusters via 4:1 concentrators. Each core comprises a processor node, an L1

data/instruction cache, a slice of a shared L2 cache, and a slice of the cache coherency directory. The cores operate on a 1.25 GHz clock. We allocate a 32KB private 4-way L1 cache to each of the 64 cores, and we allocate a 512KB private 8-way L2 cache to each cluster that is shared by the four cores in the cluster. For cache coherence, we employ a MOESI directory protocol [32]. We assume a 10mm×10mm chip, divided into sixteen 2.5mm×2.5mm clusters, with each of the four cores in a cluster occupying a 1.25mm×1.25mm area.

For the designs based on shared RETL buses, we assume designs with 32 lanes of RETLs, each lane capable of sending 20 Gb/s, which provides an aggregated throughput of 640 Gb/s. This means that without spatial partitioning, 64 bytes may be sent over the shared RETL bus per core clock period of 0.8 ns, which corresponds nicely to a cache line. With spatial partitioning into P parallel buses, 64 bytes may be sent over one of these buses in P cycles. For example, for $P = 2$, 64 bytes may be sent over one of the 2 buses in 2 cycles, or can be sent over both buses in parallel in one cycle if the source cluster has acquired both buses. The longest distance that a signal needs to travel on the bus is 15 RETL segments, or 37.5mm, which means the worst-case draining period is under two core clock cycles. We set the size of the control messages to be 8-bytes and the size of the data messages (cache lines) to be 64-bytes. These system parameters are summarized in Table 4.2.

For the dedicated interconnection architecture approach, we also assume a design with a total of 32 lanes that are split over 16 dedicated broadcast trees, so that each broadcast tree is implemented with $32/16 = 2$ lanes. With 2 lanes, 64 bytes may be sent across the broadcast tree in $32/2 = 16$ cycles. The longest distance that a signal

Table 4.2: System parameters and benchmarks.

Processor	
Core	Core clock: 1.25 GHz; #Cores: 64; Core area: 1.25mm×1.25mm
Caches	L1: private 4-way, 32KB/core; L2: private 8-way, 512KB/cluster. Coherence: MOESI (blocking)
NoC	
Network	Frequency: 20 GHz; #Lanes: 32; Shared-bus draining period: under 2 core clock cycles; Dedicated tree propagation delay: under 1 core clock cycle; Control messages: 8-bytes; Data messages: 64-bytes
Applications	
Synthetic	uniform, non-uniform
Applications	barnes (br), lu_cb (lc), lu_ncb (lnc), radix (rd), bodytrack (bt), cholesky (ck), facesim (fs), blackscholes (bs), swaptions (sw), water_nsquared (wns), radiosity (rs), raytrace (rt)

needs to travel is 6 RETL segments, or 15mm, which means that the worst-case signal propagation latency is well under one core clock cycle.

Note that for both designs based on shared RETL buses or a dedicated interconnection approach, we intentionally limit our configurations to a total of 32 lanes to stress the limits of the network capacity. As noted earlier, our designs can easily scale to many more lanes given the narrow pitch of RETLs, which could deliver better performance or support heavier workloads. For example, a design with a total of 128 lanes can achieve

an aggregated throughput of 2.56 Tb/s.

To evaluate the performance of our proposed design and arbitration schemes, we employ both synthetic and real application benchmarks. For synthetic traffic, we use the Uniform traffic model in which each cluster has equal probability of generating traffic. For real-case scenarios, we use selected applications from the PARSEC and SPLASH-2 benchmark suites. These benchmarks are also summarized in Table 4.2. To generate application traces from these benchmarks, we employ the Synfull framework [11]. The Synfull framework provides accurate trace models that account for caching behavior.

4.6.2 Performance Evaluation

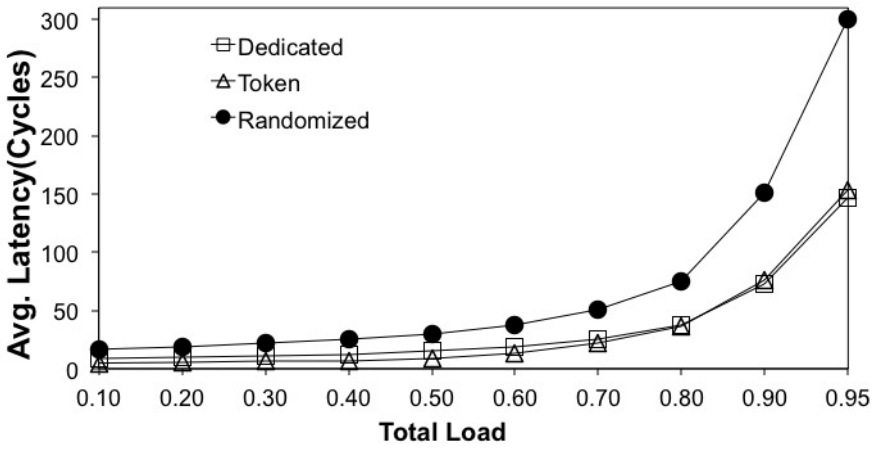
We first compare the shared RETL bus approach described in Section 4.3 with the dedicated interconnection approach described in Section 4.5. For the shared RETL bus approach, we evaluate the token-based and distributed randomized polling-based arbitration schemes described in Section 4.4. In particular, we evaluate these architectures with two traffic patterns: *uniform* and *non-uniform*. Under the uniform traffic model, each cluster i will generate a 64-byte message with probability $\lambda_i = \alpha \times 1/N$ in each cycle, where α denotes the traffic load, and 64-bytes/cycle corresponds to the aggregated throughput of the shared RETL bus. Under the non-uniform traffic model, cluster 0 and cluster 1 will each generate a 64-byte message with probability $\lambda_i = \alpha \times 1/4$ in each cycle, but the remaining clusters $i = 2, 3, \ldots, (N-1)$ will each generate a 64-byte message with probability $\lambda_i = \alpha \times 1/2(N-2)$ in each cycle. Again, α denotes the traffic load, and 64-bytes/cycle corresponds to the aggregated throughput of the shared RETL bus. The generated traffic is non-uniform in that clusters 0 and 1 will generate traffic

at a much higher rate than the other clusters. In our setup, we have $N = 16$ clusters, and we assume all traffic generated by a cluster will be destined to another cluster: i.e., all traffic is *inter-cluster* traffic that will go over the transmission line network, and the traffic generated will be queued at a FIFO at the cluster.

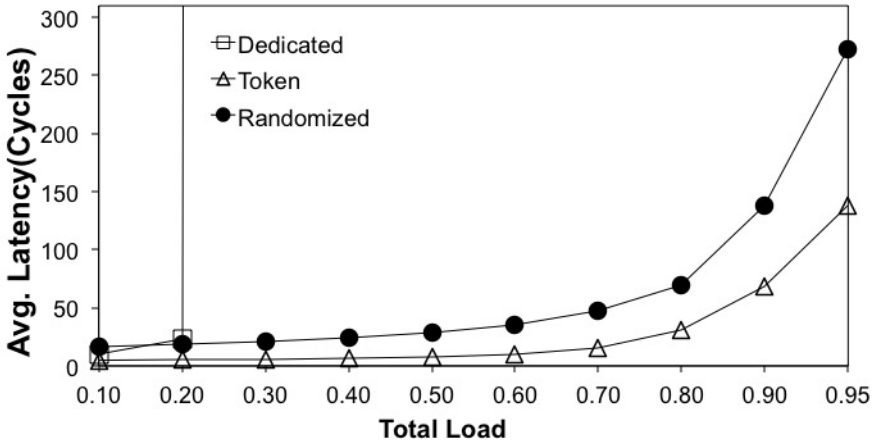
Fig. 4.11 shows the results for both uniform and non-uniform traffic. The graphs shown plot the average latency in cycles for the shared RETL approach with the two arbitration schemes and the dedicated interconnection approach with respect to the total traffic load, from $\alpha = 0.1$ (10%) to $\alpha = 0.95$ (95%). For both uniform and non-uniform traffic, we can observe from Fig. 4.11(a) and Fig. 4.11(b), respectively, that the token-based scheme performs better than the distributed randomized polling scheme for the shared RETL bus approach. Further, despite requiring a draining period of two core cycles between one transmitting cluster to another, the bus is still not yet saturated at $\alpha = 0.95$ (95%) with either arbitration scheme.

For the dedicated interconnection approach, we can observe from Fig. 4.11(a) that for uniform traffic, it also performs quite well, comparable to the token-based scheme for the shared RETL bus approach, with the dedicated interconnection network still not saturated at $\alpha = 0.95$ (95%). Recall from Section 4.5 that in the dedicated interconnection approach, each cluster has its own dedicated broadacast network that can transmit to any other cluster (or possibly to multiple clusters). Therefore, each dedicated broadacast network is equally loaded.

However, we can observe from Fig. 4.11(b) that for *non-uniform* traffic, the dedicated interconnection approach performs very poorly, saturating already after $\alpha = 0.20$ (20%). The dedicated interconnection approach actually saturates around $\alpha = 0.25$



(a) Uniform traffic.



(b) Non-uniform traffic.

Figure 4.11: Performance of the token-based and distributed randomized polling-based arbitration schemes vs. the dedicated interconnection approach.

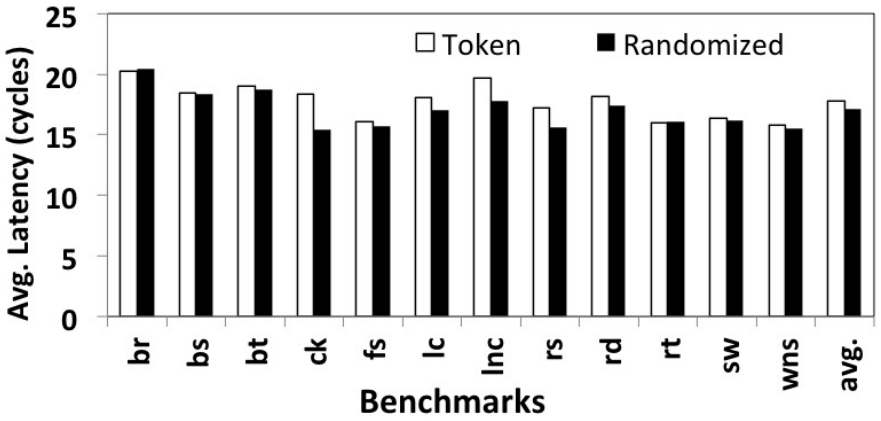


Figure 4.12: Performance of the token-based and distributed randomized polling-based arbitration schemes for real application benchmarks.

(25%), but the graph shows increments of 0.1 on the X-axis for the normalized load. The early saturation is due to the fact that each dedicated broadcast network has only $1/N^{th}$ the number of lanes as the number of lanes in the shared RETL bus approach. Thus, with a normalized load of 1.0, each dedicated broadcast network can only handle a maximum normalized rate of 1/16, for $N = 16$. However, in our non-uniform traffic model, clusters 0 and 1 generate substantially more traffic than the other clusters. In particular, clusters 0 and 1 can each generate a maximum normalized rate of 1/4, thus substantially overloading the corresponding dedicated broadcast network.

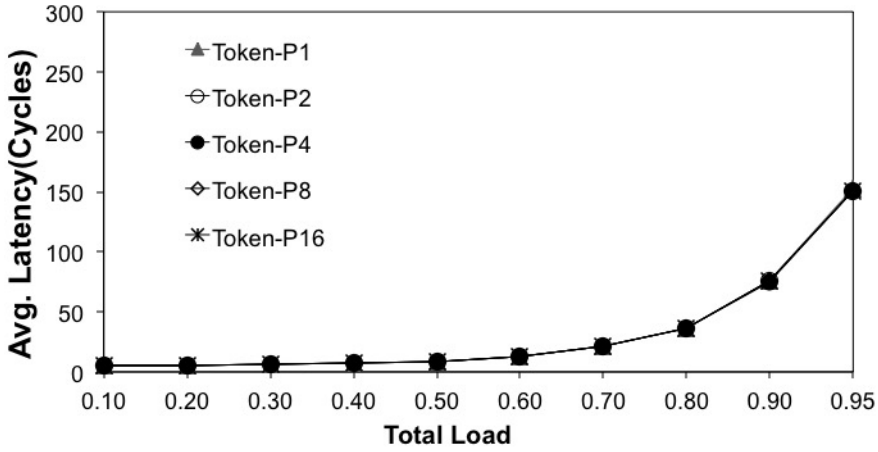
We next consider real application benchmarks from the PARSEC [14] and SPLASH-2 [75] suites. As we have already seen with the non-uniform traffic results in Fig. 4.11(b), the shared RETL bus approach performs significantly better than the dedicated interconnection approach when the traffic pattern is non-uniform (i.e., when some clusters generate more traffic than other clusters). Therefore, for the PARSEC and SPLASH-2 experiments, we only show in Fig. 4.12 the performance of the shared RETL bus ap-

proach, comparing the token-based and distributed randomized polling-based arbitration schemes. For each benchmark, we evaluated the workload for 64 parallel threads, with one thread running on each of the 64 cores. From Fig. 4.12, we can observe that both arbitration schemes have comparable performance, with the randomized polling scheme performing slightly better in most cases. In all cases, the average latency is around or below 20 cycles.

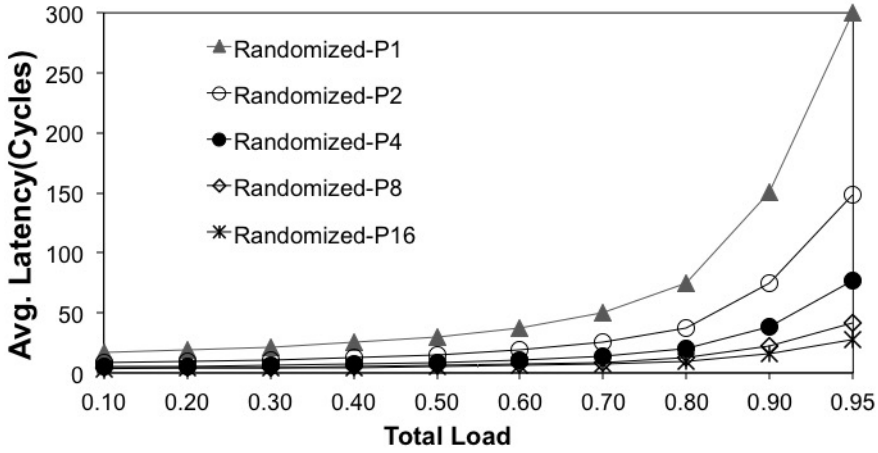
Finally, as discussed in Section 4.4.3, though the proposed arbitration schemes are effective in achieving high throughput, they only allow one cluster to transmit at a time. By having multiple parallel buses, multiple clusters can transmit *concurrently*. The spatial partitioning results for uniform and non-uniform traffic are shown in Fig. 4.13 and Fig. 4.14, respectively. As already discussed earlier, the shared RETL bus approach performs substantially better than the dedicated interconnection approach when the traffic pattern is non-uniform. Therefore, we also do not show the results of the dedicated interconnection approach in the graphs that depict the results of our spatial partitioning experiments.

In Fig. 4.13, we show the performance of the two arbitration schemes when combined with spatial partitioning for the uniform traffic model. In particular, we restrict the total number of lanes to 32 lanes in our spatial partitioning experiments. The results labeled $P1$ corresponds to just one bus with 32 lanes, $P2$ corresponds two parallel buses with 16 lanes each, $P4$ corresponds to four parallel buses with 8 lanes each, $P8$ corresponds to eight parallel buses with 4 lanes each, and $P16$ corresponds to eight parallel buses with 2 lanes each.

The spatial partitioning results for the token-based arbitration scheme are shown



(a) Spatial partitioning with token-based arbitration for uniform traffic.



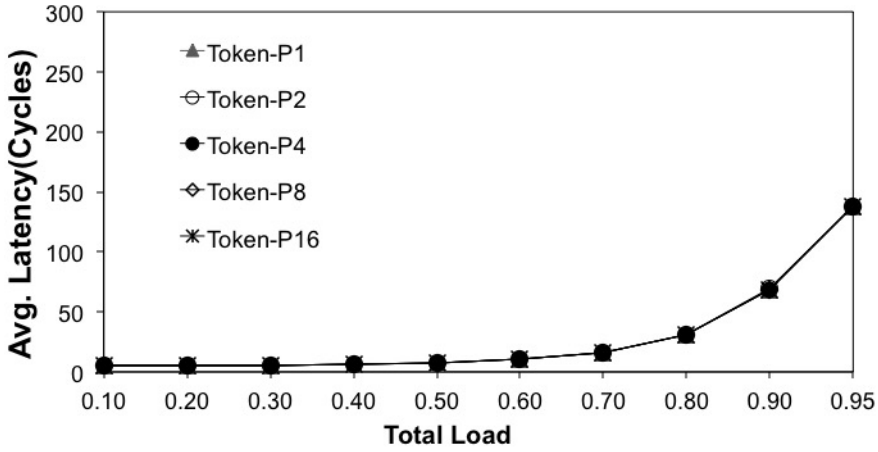
(b) Spatial partitioning with distributed randomized polling for uniform traffic.

Figure 4.13: Performance of the token-based and distributed randomized polling-based scheme for different spatial partitioning of 32 lanes under the uniform traffic model. P1 corresponds to one bus with 32 lanes. P2 corresponds to two parallel buses with 16 lanes each. P4 corresponds to four parallel buses with 8 lanes each. P8 corresponds to eight parallel buses with 4 lanes each. P16 corresponds to eight parallel buses with 2 lanes each.

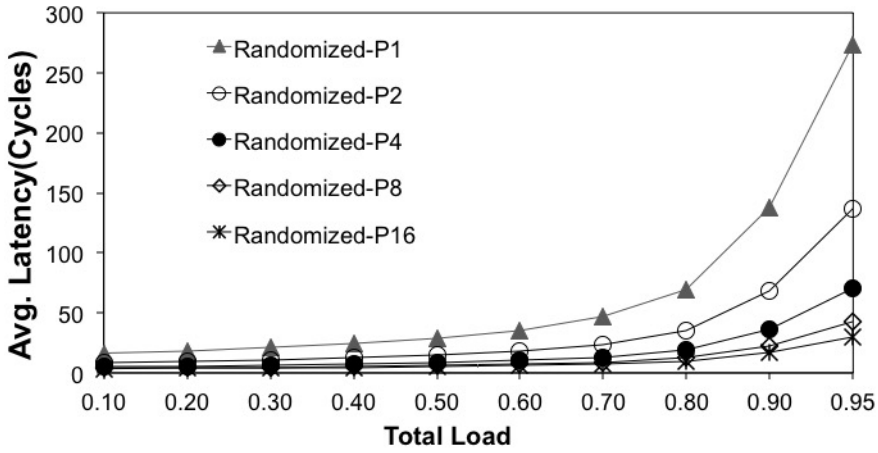
in Fig. 4.13(a). As discussed in Section 4.4.3, one problem with this approach is that clusters closer to the currently active clusters have higher priorities. If a cluster has acquired multiple buses, then the next requesting cluster downstream will likely acquire the same bundle of buses when the currently active cluster finishes. This *synchronization* problem actually leads to slightly worse performance with increasing number of parallel, but narrower buses.

On the other hand, with the randomized nature of the polling scheme, spatial partitioning actually improves performance with increasing number of parallel, but narrower buses. The spatial partitioning results for the distributed randomized polling scheme are shown in Fig. 4.13(b). As can be observed from the results, extending the distributed randomized polling method with spatial partitioning leads to better results. Even at very high loads (e.g., $\alpha = 0.95$), the average latency is very low. Effectively, the proposed randomized polling scheme achieves near ideal throughput with low latency when combined with spatial partitioning.

In Fig. 4.14, we show the performance of the two arbitration schemes when combined with spatial partitioning for the non-uniform traffic model. The results shown are again for the spatial partitioning of 32 lanes into different number of parallel buses. The spatial partitioning results for the token-based arbitration scheme and the distributed randomized polling-based scheme are shown in Fig. 4.14(a) and Fig. 4.14(b), respectively. As can be seen from these results, the same behavior can be observed as in Fig. 4.13: the token-based scheme performs better than the distributed randomized polling-based scheme without spatial partitioning. However, extending the distributed randomized polling method with spatial partitioning leads to significantly better results when the



(a) Spatial partitioning with token-based arbitration for non-uniform traffic.



(b) Spatial partitioning with distributed randomized polling for non-uniform traffic.

Figure 4.14: Performance of the token-based and distributed randomized polling-based scheme for different spatial partitioning of 32 lanes under the non-uniform traffic model. P1 corresponds to one bus with 32 lanes. P2 corresponds to two parallel buses with 16 lanes each. P4 corresponds to four parallel buses with 8 lanes each. P8 corresponds to eight parallel buses with 4 lanes each. P16 corresponds to eight parallel buses with 2 lanes each.

available lanes are partitioned into a higher number of parallel buses (four or more in our experiments).

4.7 Chapter Summary

In this chapter, we presented several designs for on-chip global communications that are based on the use of narrow-pitch repeated equalized transmission lines. These designs overcome a number of limitations associated with previously proposed designs based on on-chip transmission lines. Our designs naturally support multicast and broadcast operations and can scale to multiple terabits per seconds. In particular, we presented designs based on shared RETL buses and novel distributed arbitration schemes that can achieve high throughput and bandwidth utilization, but yet are simple to implement. In addition, we presented a design based on a dedicated interconnection architecture that can also achieve high performance.

In the next chapter, we present a fast and accurate NoC-centric simulator for evaluating NoC designs. The proposed approach accounts for complex interactions between the application, the processing cores, the memory subsystem, and the NoC.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the IEEE/ACM International Symposium on Networks-on-Chip (NOCS) 2016. Asgarieh, Yashar; Lin, Bill. The dissertation author was the primary investigator and author of this paper.

Chapter 5

NoC Simulation

Fast and accurate NoC evaluations, especially during early design stages, are essential. A CMP system has an enormous number of parameters that has to be considered in an NoC design. Every combination of these parameters can produce an entirely different on-chip communication behavior, and thus each configuration requires an independent evaluation. For a comprehensive exploration, an NoC architect needs to have access to a simulation method that allows him or her to quickly evaluate the performance of numerous system configurations to narrow down design choices for more advanced simulations in subsequent design stages. Such a simulation method must be fast and realistically model the behavior of the target application and architecture. More importantly, the simulation method should be flexible to allow easy changes to the system configuration at any level of the stack (i.e., applications, cores, memory subsystems, NoCs) and allow an NoC architect to swiftly examine the impact of a configuration on the overall performance.

While full-system simulators [16] deliver the most accurate model, they are pro-

hibitively slow to facilitate a comprehensive exploration of the design space. Alternatively, synthetic and packet-driven methods are available for quick evaluations. However, despite being fast, these approaches mostly suffer from the lack of accuracy. Therefore, neither full-system nor alternative methods simultaneously offer the agility in reconfiguration, speed in simulation, and accuracy in evaluations. As such, there is a need for a fast NoC simulation technique that can retain the agility of system reconfigurations while achieving an acceptable level of accuracy. In this chapter, we propose a cycle-accurate simulation methodology that operates at the instruction-trace level to address this goal.

5.1 Introduction

Full-system simulators are painfully slow for NoC evaluations. A common way of avoiding slow full-system simulations is to abstract the core parts away and only model the interconnection parts of a CMP system. The consequence is that such simulation methodologies cannot generate traffic through actual execution of an application; instead, they rely on reproducing the traffic artificially. There are three major ways to generate such traffic:

- *Synthetic traffic models*: In this approach, a packet generator generates traffic for different sources based on a synthetic traffic model. The synthetic traffic model used is either intended to stress some network features or model some supposedly real-case scenarios. For example, the shuffle traffic generates traffic for a source s to a destination d by using a bit-permutation function $d_i = s_{i-1} \mod b$, where b is the number bits required to encode a source address and i is the bit index. This pattern is suppose to model a traffic pattern similar to an FFT application

Table 5.1: Synthetic traffic patterns

Traffic pattern	Destination function	Target
Shuffle	$d_i = s_{i-1}$	test shuffle access patterns
Uniform	$d = random()$	test uniform distance access patterns
Bit-complement	$d_i = \overline{s_i}$	stress horizontal and vertical bisection bandwidth
Transpose	$(d_x, d_y) = (s_y, s_x)$	stress diagonal bisection bandwidth
Tornado	$(d_x, d_y) = (s_{[(x+\lvert X\rvert/2) \mod \lvert X\rvert]}, s_y)$	test dimension ordered patterns

execution[1]. Table 5.1 provides a list of example synthetic traffic patterns that are widely used for NoC evaluations. These synthetic traffic models are useful for stressing various aspects of a network design.

- *Packet trace-driven traffic*: Alternatively, packet traces can be captured by simulating a real application through a full-system simulator and recording the packet injections during simulation into a packet trace file[2]. Then, an NoC simulator can simply read those recorded packet trace files line-by-line and inject the corresponding traffic into the network at the corresponding cycles. The problem with this approach is that it does not consider complex interdependencies among consecutive packets (e.g., cache coherency request and response messages). Thus, even relatively small changes to the NoC configuration can have profound impact on the injection times of packets into the network and thus the traffic behavior. One way to mitigate this problem is to infer packet dependencies through additional full-system simulations, and then apply these inferences through post-processing

[1]However, it was shown in [11] that the actual FFT traffic behavior is quite different.
[2]Each line in a packet trace file includes injection time, source, destination, and size.

techniques [34, 55, 37, 69].

- *Synthetic coherency-aware traffic*: Another approach is to synthetically emulate the memory behavior that represents both the application and coherency traffic. [11] proposes such a method called Synfull that uses hierarchical Markov Chains to capture temporal and spatial network behaviors of an application. Like the packet trace-driven method, Synfull requires multiple full-system simulation runs for every system configuration to extract features that are necessary for generating the synthetic coherency-aware traffic models.

Among the above approaches, the *synthetic traffic* method is the least effective in modeling traffic behavior realistically. This is because realistic traffic behavior should be a function of the application, the cores, the NoC, and the memory subsystem combined, as discussed in Chapter 2. The oblivious nature of synthetic traffic models inherently cannot capture such interactions. The other two methods do attempt to take into account the system behavior by generating independent models per configuration. However, the *packet trace-driven* method still suffers from poor accuracy because it is very hard to capture the complex behaviors of CMPs at just the network level. On the other hand, the *synthetic coherency-aware* method is shown to achieve a better level of accuracy for a target architecture. However, both approaches still require time-consuming full-system simulations to generate new models on a per-application and per-configuration basis. Thus, they are not well-suited for comprehensive design space explorations.

In this chapter, we propose an NoC simulation methodology that tackles the *model generation* problem without compromising on accuracy. Our approach is based on the simulation of *instruction traces* that are produced by running applications through a

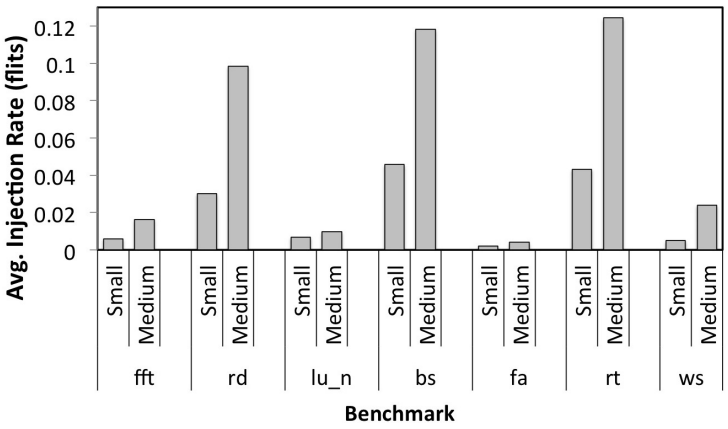
binary instrumentation tool[3]. The traces are optimized and only convey the information necessary for NoC simulations, which prevents wasting valuable CPU cycles of a host to execute simulation events that have no performance impact. Moreover, we emulate interactions of parallel application threads with an operating system. When combined with optimized instruction traces and a lightweight memory model, our approach ensures that the generated traffic to an NoC is realistic. Compared to full-system simulation, our approach is orders of magnitude faster. Compared to packet-trace and synthetic coherency-aware based methods, our approach enables accurate explorations of different system configurations without long model generation times.

The rest of this chapter is organized as follows: Section 5.2 discusses the impact of changing configurations at different levels of the system on the network traffic behavior. Section 5.3 presents a background overview on the control flow of parallel programs. Section 5.4 describes our simulation approach. Section 5.5 presents evaluation results, and Section 5.6 discusses related work. Finally, Section 5.7 concludes the chapter.

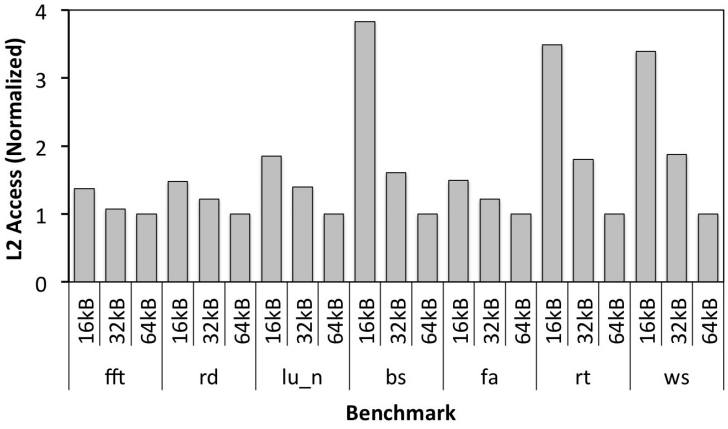
5.2 Impact of Configurations

In this section, we study the impact of configurations on the generated traffic when changing the settings of three main system configurations: dataset size, L1 cache, and L2 cache. The goal of this section is to show that the traffic produced by each combination is significantly different. In the following, we discuss the degree of traffic variation under different configurations:

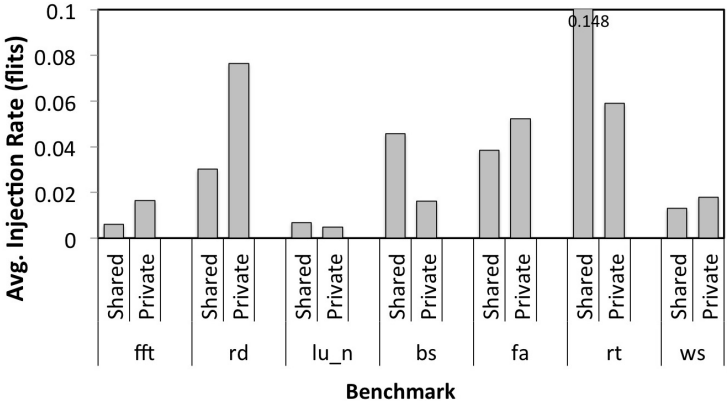
[3]Binary instrumentation allows us to get a snapshot of an application's executed instructions at an actual host speed without going through a full-system simulation run.



(a) Impact of dataset size.



(b) Impact of L1 cache.



(c) Impact of L2 cache.

Figure 5.1: The impact of different system configurations on network traffic.

- *Impact of dataset size*: Memory demands can vary substantially, depending on the size of the datasets being processed. We illustrate in Fig. 5.1(a) the impact of two different dataset sizes (small and medium) on the traffic load experienced by the network. In particular, the results are for the parallel portion of the applications with 16 threads running on a 16-core CMP with a private 32kB L1-I/D cache per tile and a shared 1MB L2 cache per tile. We see that the traffic loads are impacted differently, depending on the benchmark. Overall, the average injection rate (per flit) for the medium-size datasets can be up to 3× higher than the average injection rate for the small datasets.

- *Impact of L1 cache*: Next, we show the impact of L1 capacity on the traffic behavior. Fig. 5.1(b) depicts the total L2 accesses for three L1-I/D sizes (16kB, 32kB, 64kB). The results for each workload are normalized to the results for the private 64kB L1 caches. All other configurations are the same as above. As we can see, some of the workloads (e.g. blackscholes) are very sensitive to the L1 capacity, and their L2 accesses can increase by nearly 4× in comparison to smaller cache size configurations.

- *Impact of L2 cache*: Finally, we compare the impact of using a shared vs. a private 1MB L2 cache per tile on the average injection rate. Like the first experiment, we assume a 16-core CMP with a private 32kB L1-I/D cache per tile. As shown in Fig. 5.1(c), each L2 configuration is better for some applications, but worse for others, depending on the usage of the memory subsystem by the application. In particular, the average injection rate can vary up to 2.7× when comparing the two L2 configurations.

These results imply that changing configurations at any level of the system stack can significantly alter the traffic behavior. Thus, it requires generating an independent model for each combination. As we move towards more specialized and heterogeneous multi-processor designs to scale performance, the design space becomes much broader than the design space for traditional general-purpose architectures [24, 51, 33]. Many combinations of system configurations have to be evaluated to choose the best co-design for satisfying power, area, and performance constraints. This need for evaluating many configurations is very difficult to satisfy using current NoC simulation methodologies.

5.3 Control Flow of Parallel Applications

In Section 2.2, we explained how an instruction-initiated memory access can lead to a series of consecutive packet injections into the network. In this section, we go one step further and study the steps that an application takes for issuing an instruction on its control flow. Specifically, we first explain the execution cycle of an instruction, and then the control flow of an application thread that defines the sequence of execution cycles. Together, these two parts characterize a parallel application's network behavior.

5.3.1 Instruction Cycles

Fig. 5.2 shows the execution cycle of an instruction. It illustrates the steps that an in-order core[4] takes per instruction. As illustrated, an execution cycle starts with the fetching of an instruction from memory. The processor first searches its L1-

[4]As mentioned in Section 2.2, we assume in-order SPARC cores for all designs in this thesis.

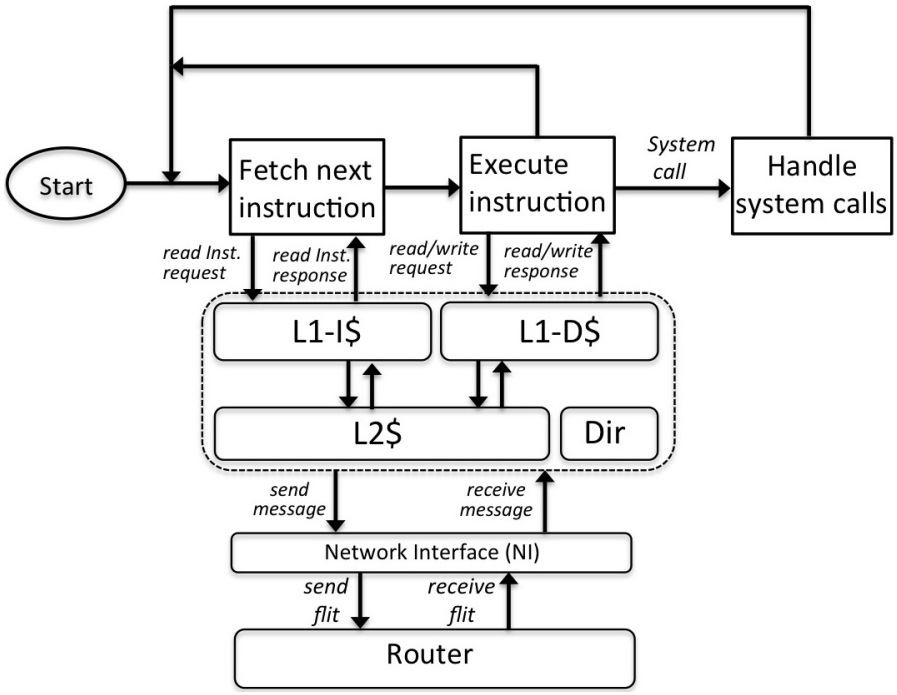


Figure 5.2: Instruction execution cycle of an application thread.

I cache for the next instruction[5]. If this fails, the processor will continue the search through the higher layers of the memory hierarchy (i.e., L2 and off-chip). We assume a RISC[6] instruction set architecture (ISA), in which instructions can be memory or ALU (Arithmetic Logic Unit) instructions. In particular, memory instructions correspond to load (read) and store (write) operations, and arithmetic instructions only operate on local registers. Instructions can also invoke a system call, which would put the current application thread on hold and move the control to the OS call handler. The application thread would remain on hold until the call is resolved and the system scheduler selects it again for execution.

[5]A Program Counter (PC) register is already loaded by the Operating System (OS) with the current instruction's address.

[6]Reduced Instruction Set Computing.

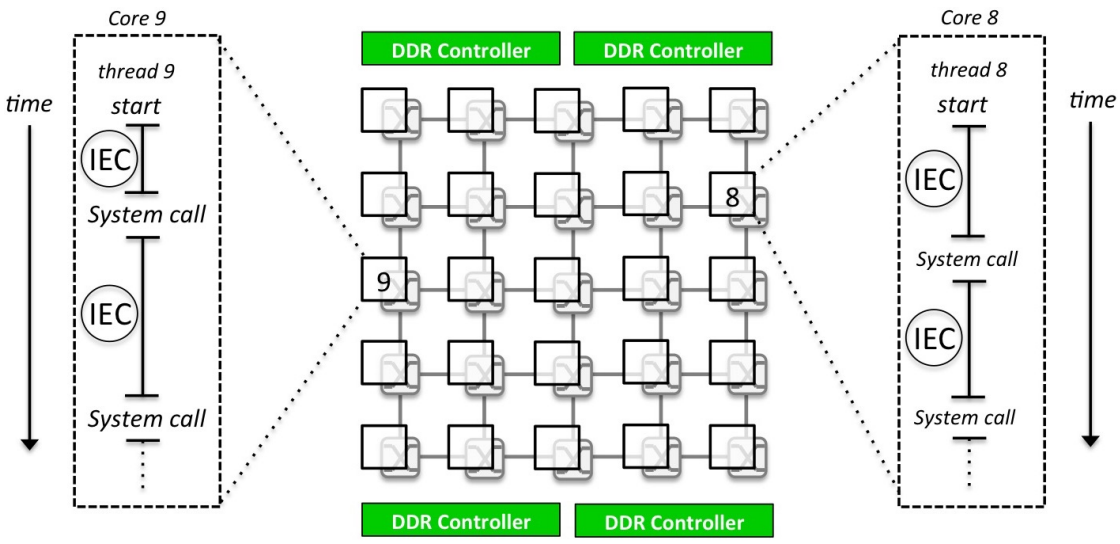


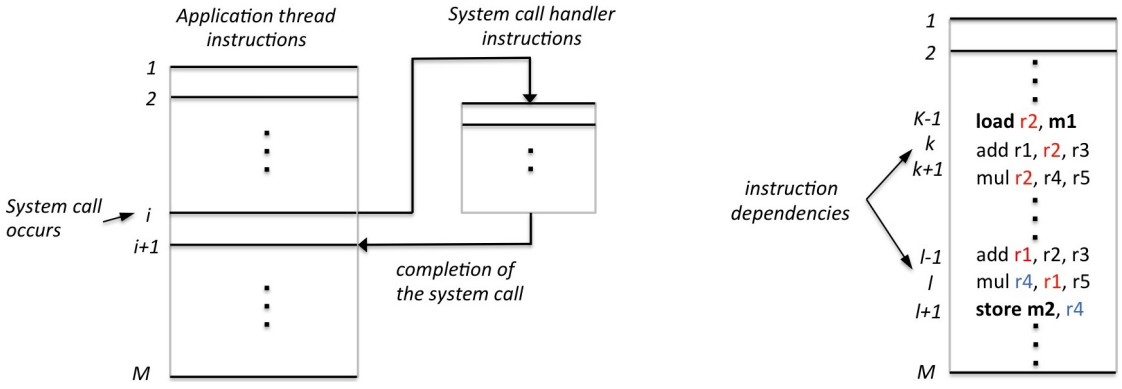
Figure 5.3: Control flow of a multi-threaded application running on a 25-core CMP.

5.3.2 Control Flow

We assume a multi-threaded program model in which threads communicate to each other via the shared memory[7]. In this parallel programming paradigm, the control flow of an application thread is defined not only by its memory accesses, but also by the points that system calls occur, which can be synchronization points like locks and barriers[8]. The instructions between following calls on the control flow of a thread can be executed independently, regardless of the execution status of other threads. We call each of these memory-bound blocks of instructions an Independent Execution Cycle (IEC). Fig. 5.3 depicts an example of two threads running on cores 8 and 9 of a 25-core CMP, with each thread looping over instructions. In this example, each thread can freely proceed by executing its instructions one-by-one until it has to stop because of a need

[7]There are other inter-process communication methods (e.g. message passing, sockets, pipes), but the shared memory model is the most widely used approach for CMP designs. We refer the reader to [74] for a detailed discussion.

[8]A lock/barrier is a synchronization method that declares a point that some/any thread or process should stop its execution cycles until some/all other threads or processes reach to the same point.



(a) System-call flow.

(b) Instruction sequence.

Figure 5.4: Instruction control flow.

to interact with the OS.

In an in-order processor, instructions are fetched, executed, and completed in the compiler generated order. Therefore, the order of memory accesses is the same as the instructions appearance on the control flow. In the case of a system call, the OS takes control of the processor core by calling a system routine to handle the request. This possibly includes a costly context-switch between the user and OS address spaces, as depicted in Fig. 5.4(a). It is worth noting that the dependencies among a group of consecutive instructions can affect the memory access sequence for an out-of-order execution model, as depicted in Fig. 5.4(b). For this chapter, we only focus on the in-order execution model and leave out-of-order models as future work.

Given the above background, we need to properly model three aspects of a multi-threaded program execution to ensure correct control flow, and hence correct traffic behavior:

- *Instruction fetch*: This aspect dictates when an instruction miss occurs at the L1-I

cache.

- *Memory access*: This aspect dictates when a memory miss occurs at the L1-D cache.

- *System call*: This aspect dictates when a thread invokes a system call and its execution is disrupted.

Next, we describe our approach to modeling these aspects.

5.4 Behavioral NoC Simulation

In this section, we introduce the behavioral NoC simulation (BNS) approach that captures the control flow, and hence the behavior of an application, from its instruction-level traces. In particular, we use a tool called Pin [50] to perform dynamic binary instrumentation (DBI). DBI leverages the host machine to generate and record instruction traces from application executions at near-native execution speeds. DBI can also be used to instrument an application to capture system calls (e.g., those used for synchronization among concurrent threads) in the generation of instruction traces. We then simulate these captured instruction traces through the memory subsystem to generate network traffic for realistic NoC evaluations. This approach captures the complex interactions between the application, the memory subsystem, and the NoC at simulation time. To make this approach more practical, we describe in this section two optimization techniques:

- *Instruction-trace reduction*: We compress the instruction trace files by filtering out arithmetic instructions that do not affect the behavior of the memory subsystem

and the NoC. This reduction leads to much smaller instruction trace files and faster simulation times.

- *Static system call handler*: We capture the system calls that occurred on the control flow of an application execution and only keep those calls where the disruption affects the traffic behavior.

In addition to these techniques, we use a cycle-accurate timing model to orchestrate all the simulator modules synchronously. Also, we implemented a lightweight memory subsystem that allows us to quickly change the memory and cache-coherence configurations to run new evaluations. We describe our approach in greater details below.

5.4.1 Instruction-trace reduction

Although DBI can generate and record instruction traces at near-native execution speeds, just a few seconds of an application execution can already generate huge trace files. To make instruction-trace simulation more practical, we propose a technique that can significantly reduce the size of instruction traces, which significantly reduces simulation times. As discussed in Section 5.3, only *memory* instructions impact packet injections. Although modeling arithmetic instructions is necessary for accurate simulations of the cores, they do not affect the performance metrics related to NoC evaluation. Therefore, we can filter out the arithmetic instructions between memory instructions.

This reduction process is depicted in Fig. 5.5. Fig. 5.5(a) shows the raw recorded instructions in their order of appearance in the application execution. Each line of this trace is a RISC arithmetic or memory instruction. For example, instruction k performs an arithmetic add operation with registers r2 and r3 as operands, with the result saved

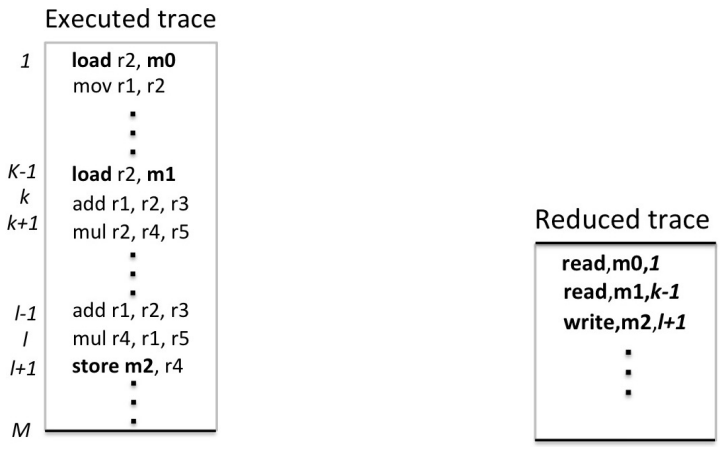


Figure 5.5: Instruction trace reduction. (a) Raw trace. (b) Reduced trace.

to r1. On the other hand, instruction $l + 1$ performs a memory **store** operation and writes the value of r4 to memory location **m2**. In the reduced trace file, as shown in Fig. 5.5(b), only the memory instructions are kept, which are represented by the type of memory operation and the memory location. For example, the first instruction [**load** r2, **m0**] in Fig. 5.5(a) is represented as [**read**, **m0**, 1] in Fig. 5.5(b) to indicate a memory **read** operation from memory location **m0** at instruction cycle 1. Similarly, the $k - 1$ instruction [**load** r2, **m1**] is represented in the second line of the reduced trace file as [**read**, **m1**, $k - 1$] to indicate a memory **read** operation from memory location **m1** at instruction cycle $k-1$, and the $l+1$ instruction [**store m2**, r4] is represented in the third line as [**write**, **m2**, $l+1$] to indicate a memory **write** operation to memory location **m2** at instruction cycle $l + 1$.

Using the reduced instruction trace file, we can simulate the memory operations line-by-line and generate the corresponding traffic through the memory subsystem and

NoC. Using the example shown in Fig. 5.5(b), the simulator first simulates the **read** operation to **m0**. After the completion of this first read operation, the simulator waits $k-2$ cycles before issuing the next **read** operation to **m1** for this thread. This is because the read operation on the first line of the reduced trace file is labeled with the instruction cycle number 1, and the read operation on the second line is labeled with the instruction cycle number $k-1$. Therefore, the simulator knows that there were $k-2$ arithmetic instructions between these two memory operations in the original trace file that were filtered out. We assume in our simulator implementation that arithmetic instructions take 1 cycle to execute. This corresponds to the $k-2$ cycles of waiting between the first two memory operations in the reduced trace file for the $k-2$ arithmetic instructions that were filtered out between them.

We see that the reduced trace is much shorter, and is hence much smaller in size, which requires a fraction of the disk space required to store the original trace. It is worthy to note that the degree of reduction depends on the memory access rate of the thread. For a multi-threaded application (as shown in Fig. 5.3), each application thread is now represented by a Reduced IEC (RIEC) instead that contains a series of memory accesses that occur between system calls. As a result, both the memory footprint of the traces and the required simulation times to simulate the reduced instruction traces are significantly reduced.

5.4.2 Static system call handler

The occurrence of a system call disrupts the control flow of an application thread and shifts the control to an OS routine, as discussed in Section 5.3. In a full-system

simulator, system calls are emulated with high-fidelity by having access to a complete system stack. However, the context switching on a system call is very expensive and can significantly slow down the simulation. Moreover, the system response time can be unpredictable, especially in many cases that involve I/O operations[9]. Alternatively, we can consider a snapshot of those calls that represent the average behavior of the system. In this way, we can eliminate these complexities and still retain an acceptable level of accuracy expected for NoC evaluations. However, similar to the discussion in Section 5.4.1, not all system calls impact the NoC. To this end, we first review the types of system interactions that can occur during the execution of a multi-threaded program:

- *Non-blocking system calls*: Non-blocking system calls are OS services in which the caller does not need to wait for a kernel notification to resume its operation. We assume the interaction between the application and OS is very fast, and that these system calls return back to user-mode immediately[10]. Therefore, we neglect the occurrence of these calls in the application's control flow.

- *Private blocking system calls*: A blocking call will stop an application's execution until the request is properly answered by the system. We assume a blocking system call is private when it only interrupts the execution of a single thread. For example, we consider I/O system calls, like reading from a file by a thread, as being private. To account for private blocking system calls, we add the estimated waiting time[11] to the RIEC instruction that occurs right after the return from the call.

[9]Many modern applications, such as client-server workloads, require having multiple network communications over TCP sockets.

[10]For example, in a non-blocking I/O, a process can submit its I/O request to a kernel buffer and immediately resumes its execution. The process will be later notified upon completion of the I/O task.

[11]We assume the waiting time corresponds to the moment that a thread goes into waiting mode until it is selected by the scheduler again and resumes execution.

- *Shared blocking system calls*: We consider a blocking system call as shared when two or more threads get affected because of the sharing a resource, which is typically a barrier or lock for a multi-threaded shared-memory application. A shared blocking call is commonly used for synchronization among threads in a multi-threaded program. Since it alters the control flow, we focus on developing techniques to emulate its behavior.

Next, we describe how we statically address the shared blocking system calls.

Implementation of shared blocking calls

We implemented a binary instrumentation tool [50] to record the shared blocking calls of a multi-threaded program. The calls are captured as they occur in a normal execution of the program at a host[12]. Then, we build a direct acyclic graph (DAG) with the calls as vertices, and the RIECs of threads as edges. That is, we replace the instruction for a shared blocking call with a corresponding vertex in the generated graph. These vertices act as synchronization points. When a thread performs all the memory accesses listed in its RIEC and reaches to this point, it will block until all other threads entering this vertex arrive.

Fig. 5.6 illustrates the proposed method for a parallel program with three threads and three synchronization points. Each thread starts to execute the memory accesses in its reduced instruction trace (as discussed in Section 5.4.1) until it blocks on the vertex V_1. If the thread is the last arrival, it will wake up the other blocked threads, and all of them can resume their normal execution[13]. A similar procedure occurs for the V_2 and

[12]Specifically, we capture the *futex_wait()*, *futex_wake()*, *futex_cmp_requeue()* system calls in a linux-based system.

[13]Note that the threads that arrive first to these points in a real execution will invoke a wait system

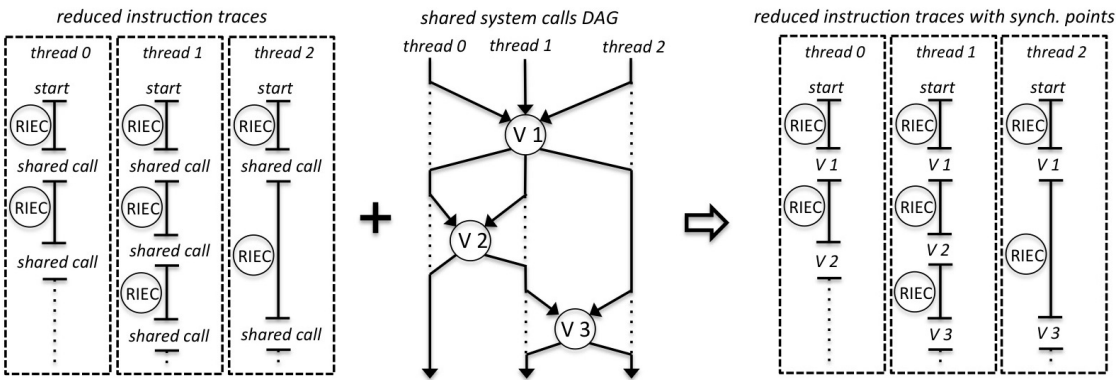


Figure 5.6: Injecting synchronization points in reduced instruction traces of application threads.

V_3 synchronization points.

To see which of the three system call classes appear more frequently, we did a study on the SPLASH[76] and PARSEC[14] benchmark suites. For CMPs, it is common to consider only the parallel part of a benchmark for evaluations as the *region of interest* (ROI). Therefore, we present results for both full and partial executions of the benchmarks. Table 5.2 shows the results for three of the benchmarks. As we can see, although all system call types occur in full runs of the benchmarks, the shared blocking system calls are the dominating type for the parallel regions. We can also see that the region of interest (the parallel part) is considerably smaller in instruction count than the full application.

Table 5.2: Application behavior at various instruction blocks (16-core, shared MOESI).

Application	Block	Inst. count	shared system calls
fft	full	3884535	59%
	ROI	917580	100%
radix	full	6595950	82%
	ROI	1677496	∼100%
lu_ncb	full	21905661	78%
	ROI	3266720	∼100%

Table 5.3: Comparison of simulation methods.

Simulation method	Approach	Memory model	Level	Model gen. cost (per combination[14])	Simulation time
full-system	Full emulation	Yes	Application	[hours,weeks]	[hours,weeks]
Synthetic	Probabilistic	No	Network	[hours,weeks]	[seconds,minutes]
Synthetic (Synfull)	Probabilistic	Approx.	Network	[hours,weeks]	[seconds,minutes]
Packet-trace	Packet-driven	No	Network	[hours,weeks]	[seconds,minutes]
Net-trace	Dependency-aware packet-driven	No	Network	[hours,weeks]	[seconds,minutes]
BNS	Abstract. emulation	Yes	Application	[seconds,minutes]	[seconds,minutes]

5.4.3 Cycle accuracy and memory model

For the NoC domain, having an accurate timing model is a necessity. We have implemented a cycle-accurate event-driven approach. That is, the simulator reads the reduced instruction traces line-by-line to schedule the next memory access based on the waiting cycles specified at each trace line in the RIEC format. We assume every arithmetic instruction takes a single CPU cycle to complete. All memory requests are serviced by a lightweight memory subsystem to determine the packets that get injected into the network. To address this issue, we utilized a two-level cache architecture for a tile-based shared memory CMP [10]. Each tile contains its local L1 instruction and data caches and a part of a shared L2 cache that is evenly distributed among the tiles. Each line of the generated reduced instruction trace then initiates a memory access and drives the memory subsystem accordingly. To maintain coherency, we adapted a directory-based MOESI protocol. Such architectural implementations, when combined with the techniques introduced in Sections 5.4.1 and 5.4.2, enable the BNS approach to achieve all the simulation reconfigurability, speed, and accuracy goals that we mentioned at the beginning of this chapter. Table 5.3 compares the BNS approaches with other simulation methodologies. As we cab see, it is the only methodology that can simultaneously satisfy all three objectives.

---

call (*futex_wait*) until another thread invokes a wakeup system call (*futex_wakeup*), at which point, they can resume execution.

5.5 Evaluation

We use gem5 [16], which is widely accepted for its accuracy, to run full-system simulations, and Garnet [8], a cycle-accurate interconnect simulator integrated into gem5 for NoC evaluation. As mentioned earlier, we use Pin [50], a dynamic binary instrumentation tool to record the control flow of applications. Both the full-system simulator and host machine use the same version of Linux kernel respectively for simulation and instrumentation. For the full-system simulator, we control the OS scheduler by fixing the threads-to-cores affinity for the tested *pthread* applications to ensure that the mapping between the application threads and simulated cores does not affect the experimental results. For all the experiments in this section, the number of application threads is equal to the number of cores. We selected applications from the SPLASH-2 [76] and PARSEC [14] benchmark suites to cover a range of traffic behaviors. The results are for the parallel part of the applications. We use a 4×4 mesh NoC for all the experiments. Table 5.4 shows the system configurations.

5.5.1 Accuracy

Fig. 5.7 shows the average latency errors, L_{error}, for BNS and Synfull [11], where L_{error} is calculated as follows[15]:

$$L_{error} = \frac{L_* - L_{full}}{L_{full}}$$

Comparing to Synfull presented in [11], BNS has lower errors across all benchmarks. As discussed in Section 2.2.2, the packets injected into the network by cache coherence are highly interdependent. Since [11] follows a statistical approach to modeling such complex

[15]In this formula, $*$ represents BNS or Synfull [11], and *full* represents full-system simulation.

Table 5.4: System Configuration

Tiles	16 cores, 2-level cache hierarchy
L1 I/D cache	32 Kbytes, 4-way, 1-cycle access latency, LRU replacement policy
LLC - L2 cache	1024Kbytes, 8-way, 5-cycle access latency, LRU replacement policy
DRAM	200-cycle access latency
Coherency protocol	MOESI
Network types	Mesh
Router	5-stage pipeline, 2 VCs, 64 bytes input/output buffer
OS	x86_64 GNU/Linux
Kernel	3.2.0-23-generic
Compiler	gcc version 4.1.2
Applications	fft, radix (rd), lu_ncb (lu_n), blackscholes (bs), fluidanimate (fa), raytrace (rt), water_spatial (ws)

interactions, it inevitably has higher errors than BNS, which truly models the memory subsystem and the complex interactions between the application, memory subsystem, and NoC. For both approaches, the degree of errors varies depending on the benchmark. For applications like *radix* that have a limited number of sharings, the coherency traffic is less complicated and only flows among a few network nodes per access. In contrast, other applications like *barnes* have a higher number of sharings, which makes their behaviors highly unpredictable. For [11], the errors are due to the statistical nature in which packet destinations are selected. On the other hand, for BNS, the errors are due to possible changes in the ordering that parallel threads access shared cache lines, which can lead to different coherency behaviors.

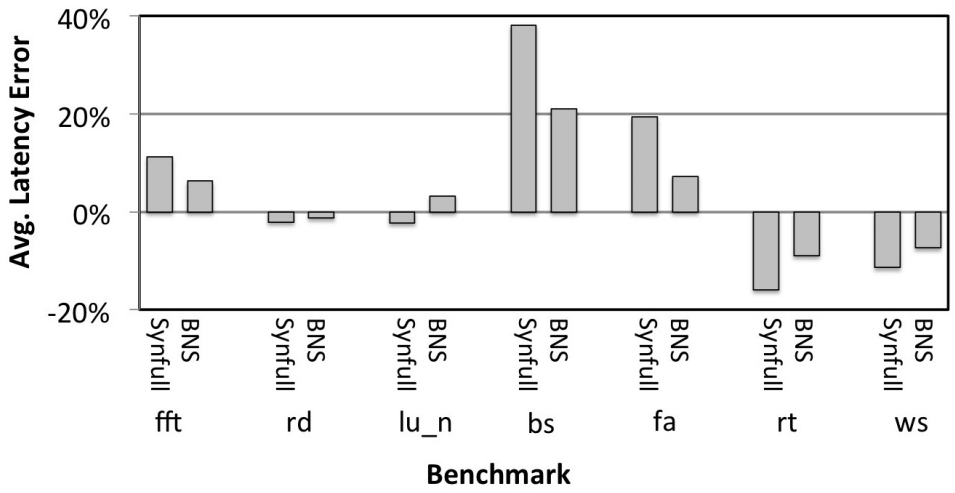


Figure 5.7: Comparison of packet latency error.

5.5.2 Performance

Simulation runtimes are shown in Fig. 5.8. The results are normalized to the runtimes of full-system simulation. Across all benchmarks, we see up to two orders of magnitude improvements for both BNS and Synfull [11]. Comparing BNS to Synfull, BNS on average has longer runtimes because of the overhead of emulating the memory subsystem. However, in contrast to Synfull, BNS can quickly generate new models without the need for time-consuming full-system simulations. Therefore, it provides a much more powerful tool for a comprehensive exploration of the design space by enabling architects to quickly evaluate different system configurations.

5.6 Related Work

A wide variety of works have been proposed as alternatives to expensive full-system simulations. Huang et al. [37] employs a bloom filter to extract packet depen-

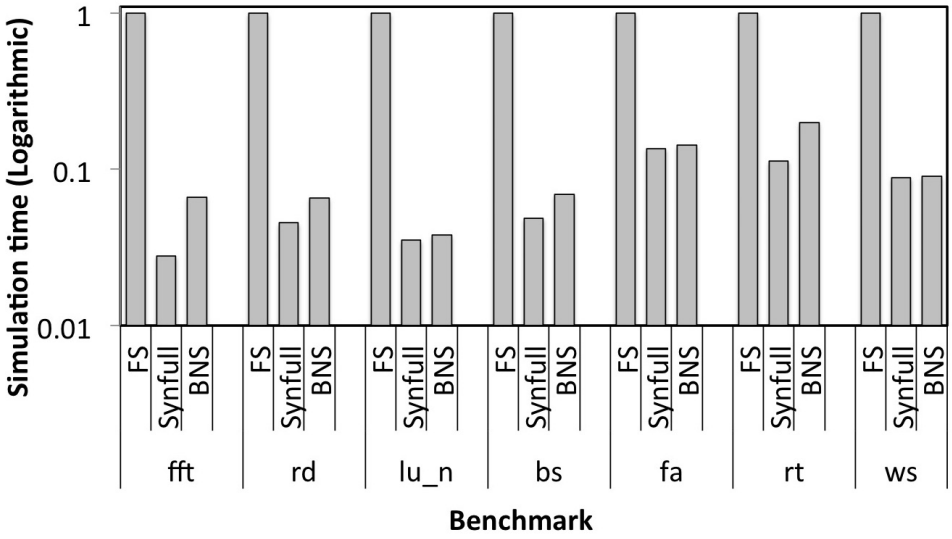


Figure 5.8: Comparison of simulation runtimes.

dencies for the case when an MPI parallel programming model is used. Hestness et al. [34] proposes to infer packet dependencies from the ordering of memory accesses. Finally, Trivio [69] proposes a methodology to keep track of packet dependencies by injecting auxiliary packets into packet traces. Despite these efforts, packet trace-driven approaches do not provide an acceptable level of accuracy. This is because realistic traffic behavior has to reflect the complex interactions between the application, the cores, the memory subsystem, and the NoC. The oblivious nature of packet trace-drive approaches inherently cannot capture such casual interactions (e.g., network packets are dynamically created in response to memory requests and cache coherence preservation). Besides, all these methods still depend on full-system simulators to generate packet-trace models.

The use of dynamic binary instrumentation (DBI) for designing scalable functional multiprocessor simulators is addressed in several works [53, 60, 39, 18]. However, the focus of these works is on the cores, not on the NoC. For example, Miller et al. [53]

achieves considerable speedup by parallelizing the multiprocessor simulation. However, their approaches loses accuracy in NoC evaluation because of the use od a loose timing model. Sanchez et al. [60] addresses the accuracy issue by employing more advanced parallelization and synchronization techniques. Despite the improvements, their approach still has to perform synchronization among parallel threads on every few instructions to achieve the best accuracy. With such frequencies of synchronization, the performance benefit of [60] diminishes – i.e., it becomes as slow as a sequential simulator.

Alternatively, synthetic traffic models can be used for fast NoC simulations [72, 65, 31]. The traffic generated by most of the basic synthetic approaches is not realistic because they typically only model the behavior of an application, not the full-system (e.g., caches, I/Os, synchronizations). To capture some aspects of the full-system, Badr and Jerger [11] propose a synthetic cache coherency-aware approach that uses hierarchical Markov Chains to generate more realistic traffic models. However, similar to the packet trace-driven methods, it too relies on expensive full-system simulations to produce traffic models.

5.7 Chapter Summary

In this chapter, we argued that current simulation methodologies are not well-suited for a comprehensive NoC design space exploration because they are very slow in either simulation or model generation. The reason is that these methods either rely on simulators with full system models or use full-system simulations to generate network traffic models. We addressed this problem by proposing the Behavioral Network Simulator (BNS), which eliminates the need for full-system simulators in any of the NoCs

evaluation phases. BNS is a cycle-accurate and trace-driven simulation approach. It is driven by instruction-level traces and realistically captures an application's network behavior without the need for slow full-system simulations. BNS employs two major ideas to achieve this goal. First, BNS employs a reduction step to compress a raw instruction trace in order to shrink its size, which in turn improves the simulation speed. Second, BNS employs a system call handling technique to preserve the control flow of a parallel program. This removes the need to simulate the operating system and secondary library modules during behavioral network simulation. In contrast to previous approaches, BNS is fast for both simulation and model-generation.

Chapter 5, in part, is in part, is currently being prepared for submission for publication of the material. Asgarieh, Yashar; Lin, Bill. The dissertation author was the primary investigator and author of this material.

Chapter 6

Conclusion

6.1 Thesis Summary

Multi-core processors are everywhere, ranging from general-purpose multi-cores [3, 1], to embedded processors [4], heterogeneous MPSoCs [6], and emerging accelerator-rich architectures [28, 33, 23]. They are used to power data centers and cloud computing, consumer applications like smartphones and tablets, and emerging applications like self-driving vehicles and Internet-of-Things. With an increasing number and diversity of cores, the on-chip network has arguably become the central performance bottleneck. In particular, long communication latencies across chip are often *the* main limiting factor in achieving higher performance or more flexible usage of on-chip resources.

Computer architects have sought to limit the impact of long on-chip latencies by avoiding long-distance data transfers. However, locality-aware approaches are becoming less and less effective. For example, tracking the state of all cache lines is becoming much more expensive with increasing network diameter. In some emerging applications, it may

be impossible or very difficult to avoid long distance communications. For example, in multi-tenant cloud computing, a multi-core processor needs to be virtualized into many virtual machines, but the allocation of on-chip resources to each virtual machine may have to come from a highly fragmented pool of resources.

In this thesis, we tackled the long on-chip latency problem *head-on* rather than avoiding it by proposing two novel NoC designs that can provide extremely low on-chip latencies for long-distance communications. The first one proposed in Chapter 3 is based on single-cycle multi-hop traversals using asynchronous repeated wires, and the second one proposed in Chapter 4 is based on the use of repeated equalized transmission lines as a global interconnect. In addition, we also proposed in Chapter 5 a fast and accurate NoC-centric simulator that accounts for complex interactions between the application, the processing cores, the memory subsystem, and the NoC.

6.2 Future Work

There are several areas in which our proposed NoC designs in Chapters 3 and 4 can be extended. First, in many shared memory cache coherence protocols, 1-to-Many and Many-to-1 communications are needed. 1-to-Many communications are useful for broadcasting or multicasting requests, and Many-to-1 communications are useful for aggregating acknowledgments and implementing barrier synchronizations. Although 1-to-M and M-to-1 communications can be achieved with M unicast packets, doing so would generate a lot more traffic, intensify contention at each hop, and increase the amount of time necessary to complete a 1-to-M or M-to-1 communication.

In the case of our SHARP NoC design described in Chapter 3, we have so far

only considered 1-to-1 traffic. However, we believe that SHARP can be easily extended to support 1-to-Many and Many-to-1 traffic as well since these extensions have already been developed for SMART [44]. For example, for 1-to-Many flows, our propagation-based SSR arbitration approach may be able to broadcast or multicast the winning SSR to multiple output links at each hop to form single-cycle multi-hop broadcast or multicast paths. For M-to-1 communications, the aggregation approach proposed in [44] is somewhat orthogonal to their SMART NoC design. We believe that their M-to-1 aggregation approach can be combined with our SHARP NoC design as well. In the case of our transmission-line based NoC designs described in Chapter 4, they already naturally support 1-to-Many communications since the shared transmission lines act as a broadcast/multicast medium. It remains an open question as to how best to extend these transmission-line based designs to support Many-to-1 traffic.

A second area of extension is to support different priority levels for different network traffic. Chapters 3 and 4 assume all traffic have the same importance or urgency. However, there are a number of applications of NoCs in which the ability to differentiate traffic would be very useful. For example, a server at Facebook's data center may service newsfeed requests as well as provide data backups to cold-storage [13]. Given that the servicing of newsfeed requests is very latency sensitive, whereas data backups are not, it may be desirable to prioritize the newsfeed traffic over the backup traffic. As another example, in the heterogeneous MPSoCs in smartphones, it may be desirable to prioritize real-time traffic like traffic to a high-definition video decoder or from a high-definition video camera to ensure a good user experience. As a last example, in emerging accelerator-rich architectures [28, 33, 23], significant amounts of data may need to be

transferred to a remote accelerator in a timely manner. Therefore, it may be desirable to prioritize that data transfer, especially if an application has to stall until the completion of the accelerator function.

In all these cases, it may be desirable to tag different packets with different levels of priorities and have the corresponding NoC prioritize packets with higher priorities for service. For our SHARP NoC design, we can extend the SSRs with a *priority* field (e.g., we can use a 3-bit priority field to encode 8 priority levels) and extend our propagation-based SSR priority arbitration scheme to select higher priority SSRs as winners. We believe that our SHARP design can easily be extended to support this type of traffic prioritization. For our transmission-line based NoC designs, we envision extending our proposed token-based and randomize polling-based arbitration schemes to support different traffic priorities. We also envision extending our spatial partitioning ideas in Chapter 4 to provide resource isolation for different traffic priority classes.

A third area of extension is to support some form of *circuit-switching* in combination with packet-switching. This type of hybrid networks can be useful in a number of settings. For example, in virtualizing a multi-core processor, it may be desirable to aggregate a fragmented pool of on-chip caches to form a larger L2 cache for a virtual machine. In emerging accelerator-rich architectures, it may be necessary to combine an ensemble of accelerators that are far apart from each other to perform a given task. In these examples, it may be desirable to reserve connectivity in the form of *circuits* between resources that are a part of a virtual machine or an accelerator ensemble.

For our SHARP NoC design, we can setup single-cycle multi-hop paths as circuits. In this case, we would still need a way to dynamically setup single-cycle multi-hop paths

for packet-switched traffic around resources that have already been reserved for circuit-switching. Since both XY and YX routing paths may be used, there may be enough path diversity to route packet-switched traffic. However, to make hybrid circuit and packet switching feasible, a number of open questions would have to be answered. At least one open question is how to allocate resources for circuit-switching to ensure that packets can still be dynamically routed between any pair of nodes without deadlocks. For our transmission-line based NoC designs, we envision employing known techniques like time-division multiplexing to reserve portions of our shared RETL buses for circuit switching. We further envision extending our proposed token-based and randomize polling-based arbitration schemes to dynamically allocate network resources around reserved time-slots. In combination with our proposed spatial partitioning techniques, we envision implementations in which we are always guaranteed that some network resources are available for packet switching at every time slot.

The above extensions are by no means exhaustive, but they give a flavor of the range of capabilities that our proposed NoC designs can provide. With these new capabilities, researchers can contemplate new ways of thinking about computer architecture. We believe that there are many exciting opportunities still ahead based on the work in this thesis.

Appendix A

Single-cycle Multi-hop Repeated Wires

In this appendix section, we review the structure of conventional and repeated on-chip wires. The goal of this section is to help the reader understand the differences between the two technologies and how an asynchronous repeated wire can send a signal over multiple hops in a single cycle.

A.1 Conventional Wires

The performance of a wire is determined by its resistance (R) and capacitance (C). Thus, a conventional wire is typically modeled as an RC network to capture its delay and noise behavior. The delay of a wire segment can be modeled by the following equation [36]:

$$D_{wire} \propto R_{gate}(C_{diff} + C_{wire} + C_{gate}) + R_{wire}(\frac{1}{2}C_{wire} + C_{gate}) \qquad \text{(A.1)}$$

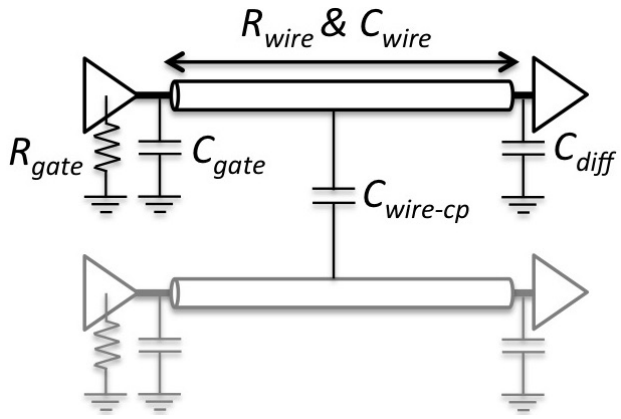


Figure A.1: The conventional wire structure.

R_{gate} and C_{gate} represent the driver's resistance and capacitance, C_{diff} represents the parasitic load, and R_{wire} and C_{wire} represent the wire's resistance and capacitance. Drivers are implemented as single/multi-stage inverters, which are sized appropriately to drive a target link. The longer the link, the bigger the driver should be to have enough fanout to drive the wire's capacitance within a given time constraint. Fig. A.1 illustrates the structure of a wire. The bandwidth of a wire can be estimated by the minimum waiting time required between successive transmissions to avoid the inter-symbol interference problem [36]. In other words, the switching frequency is limited by the time necessary for the residual current of a transition to die away.

The values of R_{wire} and C_{wire} are proportional to the wire length ℓ, which causes quadratic increases in the wire delay as Eq. (A.1) illustrates. Moreover, the spacing between adjacent wires can impact the delay because of the coupling capacitance $C_{wire-cp}$. To allow a signal to travel long distances in a single cycle, we have to address these limitations.

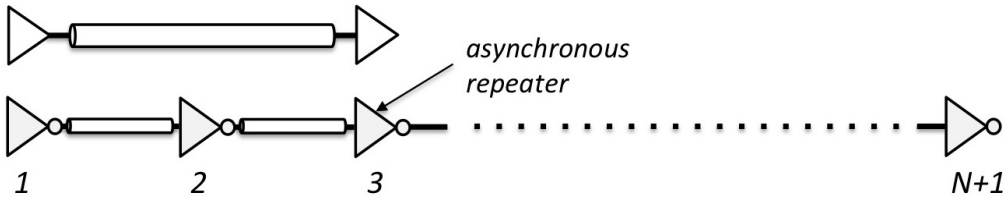


Figure A.2: The structure of a repeated wire (bottom) compare to a conventional wire (top).

A.2 Repeated Wires

Since R_{wire} and C_{wire} grow in proportion to the wire length ℓ, a standard approach to reducing wire delays is to divide a wire into smaller segments. Then, we can cascade multiple of these segments together to create a longer link, with each segment driven by a repeater. The repeaters are in turn made of inverters or buffers. The delay for this repeated structure simply grows linearly with the number of segments:

$$D_{wire}^{repeated} \propto (N \times (R_{gate}^{s}(C_{diff}^{s} + C_{wire}^{s} + C_{gate}^{s}) + R_{wire}^{s}(\frac{1}{2}C_{wire}^{s} + C_{gate}^{s}))) \qquad \text{(A.2)}$$

R_{*}^{s} and C_{*}^{s} are the resistance and capacitance values for a segment, similar to the parameters explained for Eq. (A.1). With an appropriately designed repeated structure, the quadratic increase in the original wire can be improved to a linear increase of delay with respect to the wire's length. The following parameters have to be co-optimized to achieve to the maximum performance:

- *Wire spacing*: Coupling capacitance $C_{wire-cp}$ can significantly increase the effective capacitance of a wire and can be magnified by the number of parallel wires. Thus, the spacing between adjacent wires should be enough to avoid capacitive coupling. While the absolute value of spacing can vary based on design rules, the results

in [44] suggests 3× minimum wire spacing as the optimal choice. We assume the same wire spacing for the NoC design proposed in Chapter 3.

- *Segment length*: Given the length of a segment, the maximum distance that an electrical signal can travel on a repeated link before failing timing requirements (i.e. a clock period) can be calculated using Eq. (A.2). At 45nm, the optimal segment length is around 0.4mm. However, the placement of repeaters is constrained by the tile sizes in the multi-core designs used in this thesis. Therefore, for the NoC design proposed in Chapter 3 that uses asynchronous repeated wires, we assume a segment length of 1mm to match the tile size (i.e., each tile is 1mm × 1mm in area).

- *Repeater sizing*: Repeaters are implemented as a series of inverters or buffers that are sized to adequately drive an anticipated load. A bigger repeater can drive the signal faster and thus can drive longer distances in the target time constraint, but it costs higher energy per bit. The results in [44] suggests 5× minimum inverter size as providing the best power and performance trade-offs for a 1mm segment. In this configuration, a repeated wire consumes around 26 fJ/bit per millimeter and can carry a signal up to 13mm in under 1ns. We assume the same repeater sizing for the NoC design proposed in Chapter 3.

Using the above wire spacing and repeater sizing, the propagation delay for an asynchronous repeated link is approximately 69 ps/mm.

Appendix B

Repeated Equalized Transmission Lines

In this appendix section, we first review the basic structure of unequalized on-chip TLs. Without equalization, these TLs need to have a *wide-pitch* in order to minimize the resistive loss and inter-symbol interference that can occur when transmitting a high frequency signal over a long distance. Besides occupying considerable area, these wide-pitch TLs either have limited data rates (e.g., 8 Gb/s [38]) or have to forgo multiple receivers to limit distortion [20, 19]. Alternatively, we propose to use an *equalized* on-chip TL structure [77, 78] that occupies considerably less area, and can support multicast/broadcast operations at very high data rates (e.g., 20 Gb/s). This equalized TL structure can be repeated to form longer connections.

B.1 Wide-Pitch Unequalized Transmission Lines

The use of on-chip TLs is a promising solution for high-throughput and low-power global on-chip communication [38, 40, 41]. Since the dimensions of on-chip TLs are much smaller than that of the off-chip case, on-chip TLs are resistive, which leads to lossy transmission. On-chip TLs operate in either the RC region or LC region according to the given frequency. The following equation determines the required frequency and length for an interconnect to operate in the LC region:

$$\frac{t_r}{2\sqrt{LC}} < \ell < \frac{2}{R}\sqrt{\frac{L}{C}}$$

where t_r and ℓ are the signal rising time and the length of an interconnect. According to the above equation, for a ninterconnect with lengths 1mm to 14mm, the on-chip TL can operate in the LC region at high frequency. The basic on-chip TL structure is shown in Fig. B.1. The TL structure shown is a differetial pair, which is surrounded by power and ground lines for shielding. Such differential pair structure and power/ground shielding can significantly mitigate noise. Note that TLs can also reduce the effects of process variation on latency because the signal of a TL operates in current-mode, and the latency is determined by the length rather than the width of the wire. Besides, the terminating resistor of a TL can reduce the noise induced by other interconnects. Compared to low-swing RC interconnects, TLs have greater immunity against noise.

B.2 Structure of the Repeated Equalized Transmission Lines

Unlike wide-pitch TLs, we propose to utilize equalization techniques in our TL structure in order to significantly reduce the pitch of the TLs as well as to achieve a

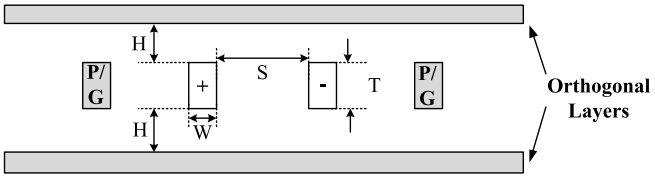


Figure B.1: The basic transmission line structure.

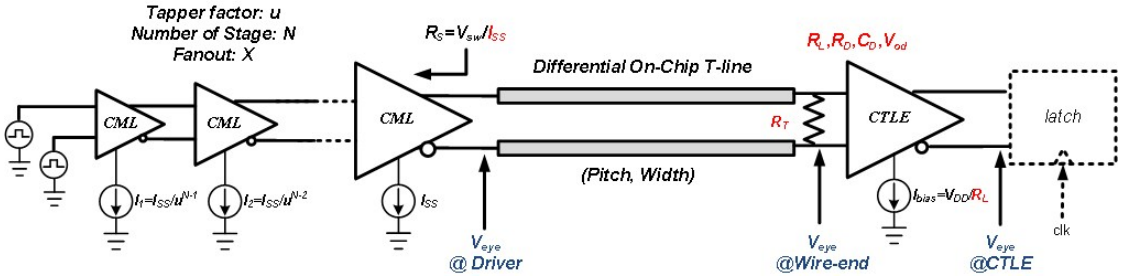
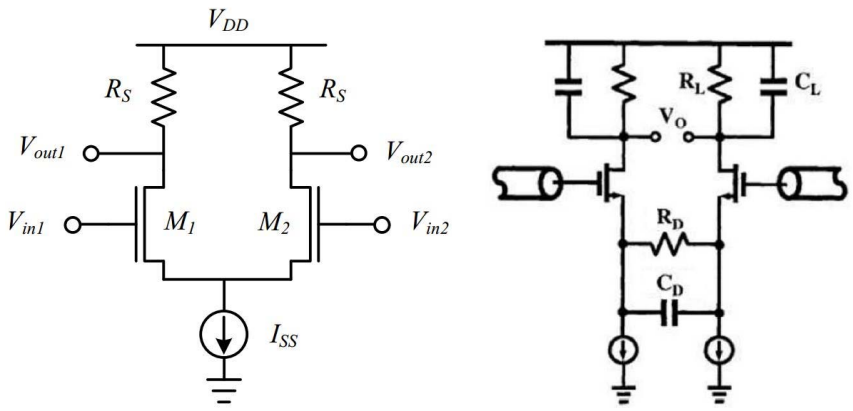


Figure B.2: The overall structure of an equalized on-chip transmission line.

high reliable data rate. In particular, we utilize the repeated equalized transmission line (RETL) design from our previous work [77, 78, 68]. Fig. B.2 shows the overall structure of a point-to-point RETL segment. The structure comprises a chain of tapered current-mode logic (CML) buffers as driver, differential on-chip TLs with terminated resistance, a continuous-time linear equalizer (CTLE) and a sense-amplifier based latch as receiver. The basic working principle is introduced as follows.

On the transmitter side (Tx), the transmitted high-speed digital signal first goes through a chain of tapered CML buffers to convert it to a low-swing differential signal, which will drive the following on-chip TL. Similar to the delay optimization of CMOS inverters or buffer chains [12], the tapered factor u and number of stages N can be decided based on the total fan-out X accordingly [35]. For a given specific driver output swing V_{sw}, the bias current I_{SS} of the final CML stage can be optimized to trade-off the driver power consumption and the eye-opening at the end of the wire. In this structure,



(a) CML (b) CTLE

Figure B.3: Schematics of CML and CTLE.

we treat the driver swing V_{sw} as a design parameter of the equalized TL and treat the bias current I_{SS} as one of the design parameters that can be optimized in the overall flow.

In terms of the on-chip global wire, we model the on-chip equalized TL by building uninterrupted differential wires that are surrounded by power and ground shielding on top of a reference ground plane, which could be a high-density lower-level metal layer as shown in Fig. B.1. We use the 2D EM Field solver [2] and a synthesized compact circuit model [43] to model and simulate the transient response of such an on-chip TL structure. The geometries (pitch, width) of the TLs are design parameters that can be tuned to adjust the characteristic impedance Z_0 and wire DC resistance to trade-off the signal attenuation with the wire area. We also add termination resistance R_T at the far-end of the TL to help improve the eye-quality after the TL [78]. The value of R_T is another design parameter to be optimized in the flow.

On the receiver side (Rx), one stage of CTLE is used to recover the transmitted

signal by boosting the eye-opening. CTLE parameters, including the load resistance R_L, source degeneration resistance R_D and capacitance C_D, and over-drive voltage V_{od}, are optimized to improve the received eye quality as well as reduce the receiver power consumption. To convert the received signals back to digital level, a dedicated synchronous sense-amplifier based latch (SA-latch) is added after the CTLE. We assume the SA-latch is a pre-designed macro in the structure. In particular, we adopt the sense amplifier design introduced in [61], and we convert it to an SA-latch by adding an SR-latch at the output.

The design guidlines of each building block have been discussed in our previous work [78], which also introduces a driver-receiver co-design methodology to determine the best set of design parameters $[I_{SS}, R_T, R_L, R_D, C_D, V_{od}]$ that can achieve the lowest energy-per-bit for the proposed equalized TL.

B.3 Co-Optimization Flow

The proposed driver-receiver co-optimization flow is illustrated in Fig. B.4. In this flow, pre-designed CML drivers and CTLE receivers are combined together as the initial solution. The co-design cost function is then estimated for certain specific solution. This stage is decomposed into three steps in the flow. First, we use HSPICE to simulate the TL step response for the specified driver resistance (I_{SS}) and termination resistance (R_T). Secondly, step response after CTLE is calculated in MATLAB by the transfer function of the CTLE [77]. Finally, the worst-case eye-opening after CTLE can be estimated using the algorithm in [63]. The co-design cost function, which will be

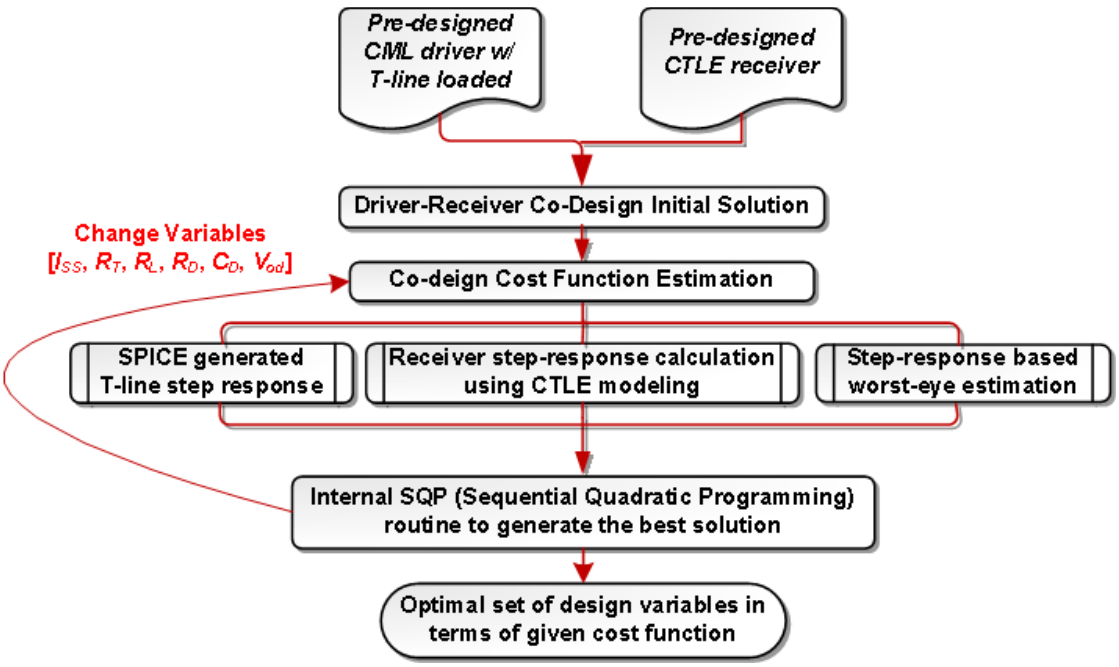


Figure B.4: The driver-receiver co-optimization flow.

minimized in the optimization, is defined as

$$f = Power + c_1 e^{c_2(V_{min} - V_{eye})} \tag{B.1}$$

where c_1 and c_2 are constant coefficients, and V_{min} is the user-defined minimal eye-opening constraint. The cost function f is used to minimize the total power dissipation of the TL segment that satisfies the minimal eye-opening constraint V_{min}. Note that the cost function excludes the delay and throughput that are given as design constraints. A non-linear optimization routine, which uses the internal Sequential Quadratic Programming (SQP) algorithm implemented in MATLAB, is called to permute the initial solution and guide the optimization iterations. In the end, the flow will generate the best solution, which is the optimal set of design parameters for the equalized TL segment $[I_{SS}, R_T, R_L, R_D, C_D, V_{od}]$ in terms of the user-defined cost function.

B.4 Co-Optimization Results

We utilize the co-optimization flow shown in Fig. B.4 to optimize the equalized TL structure with the length of 2.5mm and using the 16nm technology with the PTM model [5]. The pre-design parameters for the CML drivers $[V_{DD}, R_S, R_T, I_{SS}, V_{sw}, u, X, N]$ are $[1V, 47\Omega, 49\Omega, 6mA, 282mV, 2.5, 100, 6]$. The pre-design parameters for the CTLE receiver $[R_L, R_D, C_D, I_{bias}]$ are $[440\Omega, 110\Omega, 680fF, 1.14mA]$. In addition, the constant coefficients c_1 and c_2 are assigned 0.01 and 1, respectively. The optimal set of design variables are $I_{SS} = 1mA$, $R_T = 2.38k\Omega$, $R_L = 3.6k\Omega$, $R_D = 10\Omega$, $C_D = 0.03pF$ and $V_{od} = 37.12mV$. The width and spacing of the TL are both 1.3μm. So the pitch of TL is 2.6μm (width + space), and the width of the differential TL pair together with the power and ground shielding is 2.6×3 = 7.8μm. The targeting throughput and voltage swing are 20 Gb/s and $200mV$.

After optimization, the 2.5mm TL with our structure consumes 1.66 mW in total, where the driver takes 0.79 mW and the receiver consumes 0.87 mW. The energy per bit is 0.08 pJ/b. For the delay, the total delay is 102 ps where the TL itself requires 13 ps, the transmitter (Tx) delay is 44 ps, and the receiver (Rx) delay is 45 ps, respectively. The normalized delay is about 40 ps/mm.

Bibliography

[1] AMD Naples. *http://www.amd.com/en-us/press-releases/Pages/amd-previews-naples-2017mar07.aspx*.

[2] IBM electromagnetic field solver suite of tools. *http://www.alphaworks.ibm.com/tech/eip*.

[3] Intel Skylake. *http://www.techradar.com/news/leak-shows-an-intel-32-core-monster-cpu-to-keep-up-with-amd*.

[4] Mellanox Bluefield. *https://www.hpcwire.com/2016/06/01/mellanox-spins-ezchip-acquisition-bluefield-silicon/*.

[5] Predictive technology model (PTM). *http://ptm.asu.edu*.

[6] Qualcomm Snapdragon. *https://www.qualcomm.com/products/snapdragon/processors/835*.

[7] Agarwal, N., Krishna, T., Peh, L.-S., and Jha, N. K. Garnet: A detailed on-chip network model inside a full-system simulator. In *ISPASS* (2009), IEEE Computer Society, pp. 33–42.

[8] Agarwal, N., Krishna, T., shiuan Peh, L., and Jha, N. K. Garnet: a detailed onchip network model inside a full-system simulator. In *in Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2009*, pp. 33–42.

[9] ANSI/IEEE. Local area networks: Token ring access method and physical layer specifications, std 802.5. In *Technical report* (1989).

[10] Archibald, J., and Baer, J.-L. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst. 4*, 4 (Sept. 1986), 273–298.

[11] Badr, M., and Jerger, N. D. E. Synfull: Synthetic traffic models capturing cache coherent behaviour. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014* (2014), pp. 109–120.

[12] Bakoglu, H. *Circuits, interconnections, and packaging for VLSI.* VLSI systems series. Addison-Wesley Pub. Co., 1990.

[13] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 47–60.

[14] Bienia, C., and Li, K. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation* (June 2009).

[15] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (Aug. 2011), 1–7.

[16] Binkert, N. e. a. The gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (Aug. 2011), 1–7.

[17] Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. End to end learning for self-driving cars. *CoRR abs/1604.07316* (2016).

[18] Carlson, T. E., Heirman, W., and Eeckhout, L. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 52:1–52:12.

[19] Carpenter, A., Hu, J., Kocabas, Ö., Huang, M. C., and Wu, H. Enhancing effective throughput for transmission line-based bus. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA* (2012), pp. 165–176.

[20] Carpenter, A., Hu, J., Xu, J., Huang, M. C., and Wu, H. A case for globally shared-medium on-chip interconnect. In *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA* (2011), pp. 271–282.

[21] Chen, C.-S. O., Park, S., Krishna, T., Subramanian, S., Chandrakasan, A. P., and Peh, L.-S. Smart: A single-cycle reconfigurable noc for soc applications. In *Proceedings of the Conference on Design, Automation and Test in Europe* (San Jose, CA, USA, 2013), DATE '13, EDA Consortium, pp. 338–343.

[22] Chen, X., and Jha, N. K. Reducing wire and energy overheads of the smart noc using a setup request network. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems PP*, 99 (2016), 1–14.

[23] Chien, A. A., Thanh-Hoang, T., Vasudevan, D., Fang, Y., and Shambayati, A. 10x10: A case study in highly-programmable and energy-efficient heterogeneous federated architecture. *SIGARCH Comput. Archit. News 43*, 3 (Dec. 2015), 2–9.

[24] Coates, A., Huval, B., Wang, T., Wu, D. J., Ng, A. Y., and Catanzaro, B. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (2013), ICML'13, JMLR.org, pp. III–1337–III–1345.

[25] Dally, W. J., and Towles, B. Route packets, not wires: On-chip inteconnection networks. In *Proceedings of the 38th Annual Design Automation Conference* (New York, NY, USA, 2001), DAC '01, ACM, pp. 684–689.

[26] Dennard, R. H., Gaensslen, F. H., nien Yu, H., Rideout, V. L., Bassous, E., Andre, and Leblanc, R. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Solid-State Circuits Society Newsletter 12*, 1 (1974), 38–50.

[27] Duato, J., Yalamanchili, S., and Lionel, N. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[28] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Power limitations and dark silicon challenge the future of multicore. *ACM Trans. Comput. Syst. 30*, 3 (Aug. 2012), 11:1–11:27.

[29] Goresky, M., and Klapper, A. Fibonacci and galois representations of feedback-with-carry shift registers. *IEEE Transactions on Information Theory 48*, 11 (2002), 2826–2836.

[30] Gratz, P., Grot, B., and Keckler, S. W. Regional congestion awareness for load balance in networks-on-chip. In *14th International Conference on High-Performance Computer Architecture (HPCA-14 2008), 16-20 February 2008, Salt Lake City, UT, USA* (2008), pp. 203–214.

[31] Gratz, P. V., and Keckler, S. W. Realistic workload characterization and analysis for networks-on-chip design, 2010.

[32] Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A. Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 184–195.

[33] Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A. Toward dark silicon in servers. *IEEE Micro 31*, 4 (July 2011), 6–15.

[34] Hestness, J., Grot, B., and Keckler, S. W. Netrace: Dependency-driven trace-based network-on-chip simulation. In *Proceedings of the Third International Workshop on Network on Chip Architectures* (New York, NY, USA, 2010), NoCArc '10, ACM, pp. 31–36.

[35] Heydari, P., and Mohanavelu, R. Design of ultrahigh-speed low-voltage cmos cml buffers and latches. *IEEE Transactions on VLSI Systems 12*, 10 (2004), 1081–1093.

[36] Ho, R. On-chip wires: Scaling and efficiency. In *PhD Thesis* (August 2003), Stanford University.

[37] Huang, Y. S.-C., Chang, Y.-C., Tsai, T.-C., Chang, Y.-Y., and King, C.-T. Attackboard: a novel dependency-aware traffic generator for exploring noc design space. In *DAC* (2012), P. Groeneveld, D. Sciuto, and S. Hassoun, Eds., ACM, pp. 376–381.

[38] Ito, H., Kimura, M., Miyashita, K., Ishii, T., Okada, K., and Masu, K. A bidirectional- and multi-drop-transmission-line interconnect for multipoint-to-multipoint on-chip communications. *IEEE Journal of Solid-State Circuits 43*, 4 (April 2008), 1020–1029.

[39] Jaleel, A., Cohn, R. S., keung Luk, C., and Jacob, B. Cmpsim: A pin-based on-the-fly multi-core cache simulator, 2008.

[40] Jose, A., Patounakis, G., and Shepard, K. Near speed-of-light on-chip interconnects using pulsed current-mode signalling. In *IEEE Symposium on VLSI Circuits* (2005), pp. 108–111.

[41] Jose, A., and Shepard, K. Distributed loss-compensation techniques for energy-efficient low-latency on-chip communication. *IEEE Journal of Solid-State Circuits* (2007), 1415–1424.

[42] Jung, J. W., and Razavi, B. A 25-gb/s 5-mw CMOS cdr/deserializer. *J. Solid-State Circuits 48*, 3 (2013), 684–697.

[43] Kopcsay, G. V., Krauter, B., Widiger, D., Deutsch, A., Rubin, B. J., and Smith, H. H. A comprehensive 2-d inductance modeling approach for vlsi interconnects: Frequency-dependent extraction and compact circuit model synthesis. *IEEE Transactions on VLSI Systems 10*, 6 (2002), 695–711.

[44] Krishna, T. Enabling dedicated single-cycle connections over a shared network-on-chip. In *PhD Thesis* (February 2014), MIT.

[45] Krishna, T., Chen, C. O., Kwon, W., and Peh, L. Breaking the on-chip latency barrier using SMART. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013* (2013), pp. 378–389.

[46] Krishna, T., Chen, C. O., Kwon, W., and Peh, L. Smart: Single-cycle multihop traversals over a shared network on chip. *IEEE Micro 34*, 3 (2014), 43–56.

[47] Kumar, A., Kundu, P., Singh, A. P., Peh, L.-S., and Jha, N. K. A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator in 65nm cmos. In *ICCD* (2007), IEEE, pp. 63–70.

[48] Kumar, A., Peh, L.-S., and Jha, N. K. Token flow control. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2008), MICRO 41, IEEE Computer Society, pp. 342–353.

[49] Kumar, A., Peh, L.-S., Kundu, P., and Jha, N. K. Express virtual channels: Towards the ideal interconnection fabric. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, ACM, pp. 150–161.

[50] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[51] Magaki, I., Khazraee, M., Gutierrez, L. V., and Taylor, M. B. Asic clouds: Specializing the datacenter. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Piscataway, NJ, USA, 2016), ISCA '16, IEEE Press, pp. 178–190.

[52] Matsutani, H., Koibuchi, M., Amano, H., and Yoshinaga, T. Prediction router: Yet another low latency on-chip router architecture. In *HPCA* (2009), IEEE Computer Society, pp. 367–378.

[53] Miller, J. E., Kasture, H., Kurian, G., III, C. G., Beckmann, N., Celio, C., Eastep, J., and Agarwal, A. Graphite: A distributed parallel simulator for multicores. In *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India* (2010), pp. 1–12.

[54] Mullins, R., West, A., and Moore, S. Low-latency virtual-channel routers for on-chip networks. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2004), ISCA '04, IEEE Computer Society, pp. 188–.

[55] Nitta, C., Farrens, M., Macdonald, K., and Akella, V. Inferring packet dependencies to improve trace based simulation of on-chip networks. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip* (New York, NY, USA, 2011), NOCS '11, ACM, pp. 153–160.

[56] Park, S., Krishna, T., Chen, C.-H. O., Daya, B. K., Chandrakasan, A., and Peh, L.-S. Approaching the theoretical limits of a mesh noc with a 16-node chip prototype in 45nm soi. In *DAC* (2012), P. Groeneveld, D. Sciuto, and S. Hassoun, Eds., ACM, pp. 398–405.

[57] Peh, L.-S., and Dally, W. J. A delay model and speculative architecture for pipelined routers. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2001), HPCA '01, IEEE Computer Society, pp. 255–.

[58] Ramanujam, R. S., and Lin, B. Destination-based adaptive routing on 2d mesh networks. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (New York, NY, USA, 2010), ANCS '10, ACM, pp. 19:1–19:12.

[59] Roberts, L. G. Aloha packet system with and without slots and capture. *SIGCOMM Comput. Commun. Rev. 5*, 2 (Apr. 1975), 28–42.

[60] Sanchez, D., and Kozyrakis, C. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th annual International Symposium in Computer Architecture (ISCA-40)* (June 2013).

[61] Schinkel, D., Mensink, E., Klumperink, E., Tuiji, E., and Nauta, B. Design methodology latch-type voltage sense amplifier with 18ps setup+hold time. In *IEEE International Solid-State Circuits Conference* (2007), pp. 314–316.

[62] Seo, D., Ali, A., Lim, W.-T., Rafique, N., and Thottethodi, M. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2005), ISCA '05, IEEE Computer Society, pp. 432–443.

[63] Shi, R., Yu, W., Zhu, Y., Kuh, E. S., and Cheng, C. K. Efficient and accurate eye diagram prediction for high speed signaling. In *IEEE International Conference on Computer-Aided Design* (2008), pp. 655–661.

[64] Sorin, D. J., Hill, M. D., and Wood, D. A. *A Primer on Memory Consistency and Cache Coherence*, 1st ed. Morgan & Claypool Publishers, 2011.

[65] Soteriou, V., Wang, H., and Peh, L.-S. A statistical traffic model for on-chip interconnection networks. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation* (Washington, DC, USA, 2006), MASCOTS '06, IEEE Computer Society, pp. 104–116.

[66] Sullivan, H., and Bashkow, T. R. A large scale, homogeneous, fully distributed parallel machine, i. In *Proceedings of the 4th Annual Symposium on Computer Architecture* (New York, NY, USA, 1977), ISCA '77, ACM, pp. 105–117.

[67] Sun, C., Chen, C.-H. O., Kurian, G., Wei, L., Miller, J., Agarwal, A., Peh, L.-S., and Stojanovic, V. Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip* (Washington, DC, USA, 2012), NOCS '12, IEEE Computer Society, pp. 201–210.

[68] Sun, G., Weng, S., Cheng, C., Lin, B., and Zeng, L. An on-chip global broadcast network design with equalized transmission lines in the 1024-core era. In *International Workshop on System Level Interconnect Prediction, SLIP '12, San Francisco, CA, USA, June 3, 2012* (2012), pp. 11–18.

[69] Trivio, F., Andujar, F. J., Alfaro, F. J., Snchez, J. L., and Ros, A. Self-related traces: An alternative to full-system simulation for nocs. In *HPCS* (2011), W. W. Smari and J. P. McIntire, Eds., IEEE, pp. 819–824.

[70] Vantrease, D., Binkert, N., Schreiber, R., and Lipasti, M. H. Light speed arbitration and flow control for nanophotonic interconnects. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 304–315.

[71] Vantrease, D., Schreiber, R., Monchiero, M., McLaren, M., Jouppi, N. P., Fiorentino, M., Davis, A., Binkert, N., Beausoleil, R. G., and Ahn, J. H. Corona: System implications of emerging nanophotonic technology. In *International Symposium on Computer Architecture* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 153–164.

[72] Varatkar, G. V., and Marculescu, R. On-chip traffic modeling and synthesis for mpeg-2 video applications. *IEEE Trans. Very Large Scale Integr. Syst. 12*, 1 (Jan. 2004), 108–119.

[73] Weng, S., Zhang, Y., Buckwalter, J. F., and Cheng, C. Energy efficiency optimization through codesign of the transmitter and receiver in high-speed on-chip interconnects. *IEEE Trans. VLSI Syst. 22*, 4 (2014), 938–942.

[74] Wilkinson, B., and Allen, M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[75] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 1995), ISCA '95, ACM, pp. 24–36.

[76] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 1995), ISCA '95, ACM, pp. 24–36.

[77] Zhang, Y., Buckwalter, J., and Cheng, C.-K. High-speed and low-power on-chip global link using continuous-time linear equalizer. *Electrical Performance of Electronic Packaging and Systems (EPEPS), 2010 IEEE 19th Conference on, vol., no., pp.5-8, 25-27 Oct. 2010* (2010).

[78] Zhang, Y., Zhang, L., Deutsch, A., Katopis, G., Dreps, D., Buckwalter, J., Kuh, E., and Cheng, C.-K. Design methodology of high performance on-chip global interconnect using terminated transmission-line. *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design, vol., no., pp.451-458, 16-18 March 2009* (2009).

[79] Zhao, L., and et al. Performance, area and bandwidth implications on large-scale cmp cache design. In *In Proceedings of the Work. on Chip Multiprocessor Memory Systems and Interconnects* (2007).

[80] Zhuravlev, S., Saez, J. C., Blagodurov, S., Fedorova, A., and Prieto, M. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv. 45*, 1 (Dec. 2012), 4:1–4:28.