SC^{2}EPTON: High-Performance and Scalable, Low-Power and Intelligent, Ordered Mesh On-Chip Network

by

Bhavya Kishor Daya

BSEE, BSCEN, University of Florida (2009)
M.Eng, University of Florida (2009)

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy
at the
Massachusetts Institute of Technology
September 2015

Signature of Author .... Signature redacted
Department of Electrical Engineering and Computer Science
August 21, 2015

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

Certified by .... Signature redacted ....
Anantha P. Chandrakasan
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .... Signature redacted ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science,
Chair, EECS Committee on Graduate Students

SC^{2}EPTON: High-Performance and Scalable, Low-Power and Intelligent, Ordered Mesh On-Chip Network

by

Bhavya Kishor Daya

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2015, in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

ABSTRACT

Over the last few decades, hindrances to performance and voltage scaling led to a shift from uniprocessors to multicore processors, to the point where the on-chip interconnect plays a larger role in achieving the desired performance and power goals. Shared memory multicores are subject to data sharing concerns as each processor computes on data locally, and needs to be aware of accesses by other cores. Hardware cache coherence addresses the problem, and provides superior performance to software-implemented coherence, but is limited within practical constraints, *i.e.* area, power, timing. Scaling coherence to higher core counts, presents challenges of unscalable storage, high power consumption, and increased on-chip network traffic.

SC^{2}EPTON targets the three challenges with three on-chip networks - SCORPIO, SCEPTER, SB^2. SCORPIO addresses the unscalable storage plaguing directory-based coherence, with a 36-core chip prototype showcasing a novel distributed global ordering mechanism to support snoopy coherence over scalable mesh networks. Although the downsides of a directory are averted, the network itself consumes a significant fraction of the total chip power, of which the router buffer power dominates. SCEPTER is a bufferless mesh NoC that reduces the network power consumption, and achieves high performance by intelligently prioritizing, routing, and throttling flits to maximize opportunities to bypass on dynamically set, virtual single-cycle express paths. For unicast communication, SCEPTER performs on-par with state-of-the-art buffered networks, however broadcasts exacerbate the link contention at bisection and ejection links, limiting performance gains. SB^2 addresses the broadcast traffic in bufferless NoCs with a TDM-based embedded ring architecture that dynamically determines ring access, allows multiple sources simultaneous contention-free access, and sets the control path locally at each node within the same cycle. The three NoCs contribute key elements to the SC^{2}EPTON architecture, resulting in a low-power and high-performance bufferless snoopy coherent mesh network.

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

Thesis Supervisor: Anantha P. Chandrakasan
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

To the Almighty for intelligence and inspiration.

To family for their continued guidance, encouragement and support.

To Professor Li-Shiuan Peh, Professor Anantha Chandrakasan and Professor Srinivas Devadas for their guidance and support during the many years of research exploration and its culmination with this thesis.

To LSP Group Members (Chia-Hsin Owen Chen, Suvinay Subramanian, Woo-Cheol Kwon, Sunghyun Park and Tushar Krishna) for their collaboration on the development of the Scorpio Chip.

To one and everyone at MIT and EECS who may have played a role, large or small, in my success : administrative assistants (Maria Rebelo, Mary McDavitt), fellow students, faculty members, counsellors, graduate office members (Janet Fischer), thesis advisors, research committee members, and last but not the least CSAIL, MTL and LSP group members.

To the various service providers of CSAIL and TIG for my day-to-day needs.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The computing industry rapidly progressed over the last few decades, pervading every aspect of life today. From mobile and embedded systems to servers, integrated circuits are tailored to provide continued performance growth. A multitude of innovations in architecture, circuits, devices and fabrication technologies converged to tackle the performance, area, and power challenges of today. Gordon Moore, co-founder of Intel and Fairchild Semiconductor, observed a trend, aptly named in retrospect as *Moore's Law*, where the number of transistors in an integrated circuit approximately doubles every 18 months or 1.5 years. Moore's Law set the pace for the computing industry and coupled with transistor scaling rules such that an exponential increase in performance was achieved each technology generation, over the last few decades. Robert Dennard identified Metal-Oxide Semiconductor Field Effect Transistor (MOSFET) and circuit parameters for providing systematic transistor improvements when scaling. [29] Reducing the minimum feature size by $0.7\times$, each process technology generation, provides a $2\times$ increase in transistor density. In the early years, 1970s and 1980s, a new technology generation occurred every 3 years. Combined with an increase in die area, the transistor count grew $4\times$ every 3 years or as Moore observed, transistor count doubling every 18 months. Figure 1-1 shows the clock frequency and transistor count, for Intel microprocessors over the last three decades. The clock frequency steadily increased until year 2010, and dropped thereafter as the computing industry encountered a critical point - transistor scaling limits end the golden age of scaling, such that performance and power scaling is not as easily achieved.

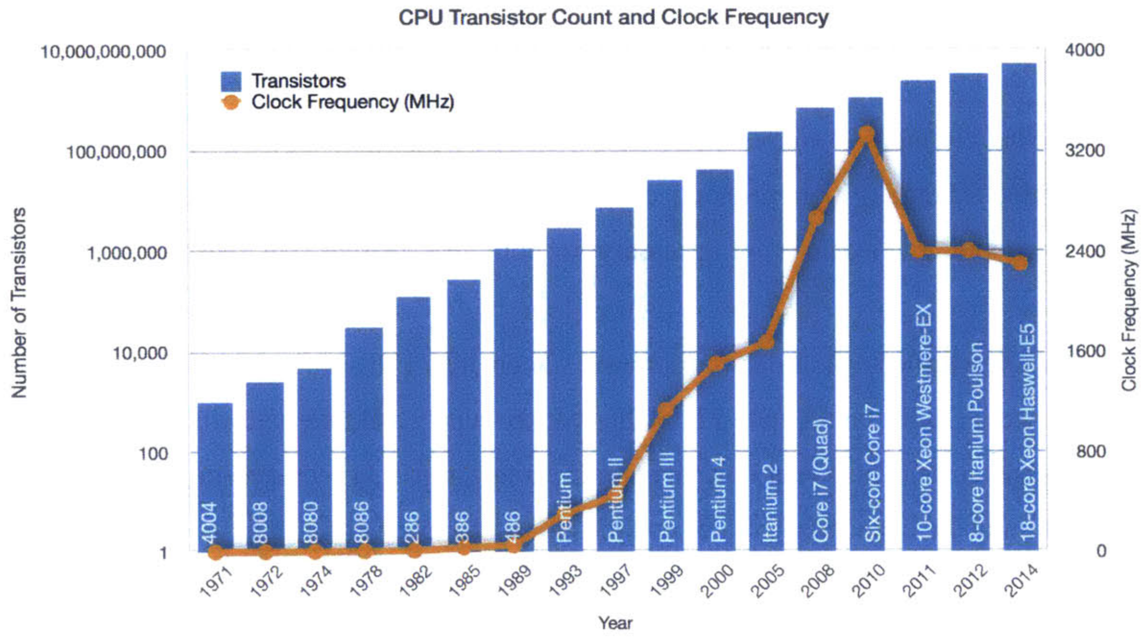


Figure 1-1: Moore's Law and Slowing Clock Frequency Scaling

The Dennard scaling rules ensured that the total power remains constant when switching to the next process generation. Table 1.1 shows Dennard's MOSFET scaling rules, where k is a unitless scaling constant, and the electric field remains constant as the features scale. When scaling with a factor of $k = sqrt(2)$, feature size reduces by $1/k$, transistor count doubles, and frequency increases by 40% every 1.5 to 2 years. The power consumption of complementary MOS (CMOS) chips can be modeled as in Equation 1.1, which is the sum of dynamic power and leakage power consumption, where Q is the number of transistors, f is the clock frequency, C is the capacitance, V is the supply voltage, and $I_{leakage}$ is the leakage current.

$$P = Q * f * C * V^2 + V * I_{leakage} \tag{1.1}$$

Each generation the power scales as $P = 2 * k * (1/k) * (1/(k^2))$, assuming leakage is negligible, thus constant power scaling is achieved when $k = sqrt(2)$. The key to maintaining the constant electric field and power density is voltage scaling, however recently it has reached its lower limit. Dennard assumed the transistor threshold voltage(V_T) would continue to scale by $1/k$, but neglected the impact of subthreshold leakage on the chip power consumption. His assumption held true until 65 nm technologies were reached. At which point the subthreshold leakage consumed a significant portion of the total power, prompting concerns on power scaling, and leading to numerous research on low-power circuits and architectures. Power limits the gate oxide thickness(t_{ox}) scaling as well, as gate oxide leakage is more pronounced with thinner dielectrics. Hence, Dennard scaling rules break down and new avenues are necessary for continued feature size, performance and power scaling.

While remarkable breakthroughs in strained silicon, high-k dielectrics and 3D FinFET transistors have allowed leakage control up to 22 nm thus far, the supply and threshold voltage remain constant and no longer scale. The power thus increases by a factor of 1.33 each generation for the same die area, assuming the capacitance scales and the clock frequency is constant. Feature size reduction still increases the transistor density each generation, however the clock frequency has hit a wall, and scaling has slowed. Single chip performance does not improve if the clock speed is constant. Instead the computational throughput can be increased if multiple processors are working in parallel, giving rise to the many-core revolution. Architecting many-core chips involves the delicate challenge of balancing physical constraints, area, timing and power while achieving high performance.

1.1 Many-Core Revolution

In addition to the performance improvements of the single-core processors that arise from clock frequency scaling, Instruction-Level Parallelism (ILP) provides performance benefits by increasing the amount of work performed each cycle. However, there are three limits to performance gains: ILP wall, memory wall and power wall. Computer engineers can no longer rely on the clock frequency scaling due to the power wall, serial performance acceleration due to the ILP wall, and low memory access latencies due to the memory wall.

ILP Wall. To yield improved serial performance of single-core processors, the key idea of ILP is to increase parallelism with regard to the instructions processed. Increasing the word length and number of instructions executed per cycle has been effective at scaling the performance. Duplicated hardware speculatively executes future instructions before the results of current instructions are known and provides safeguards to prevent data corruption errors caused by out of order execution.

Table 1.1: Dennard Scaling

MOSFET	
Dimensions (t_{ox}, L, W)	$1/k$
Doping Concentration (N_a)	k
Supply Voltage	$1/k$
Threshold Voltage	$1/k$
ON Current	$1/k$
Effective Resistance	1
Capacitance	$1/k$
Gate Delay	$1/k$
Clock Frequency	k
Power per Gate	$1/k^2$
Area per Gate	$1/k^2$
Interconnect	
Dimensions (W, s, h)	$1/k$
Wire Resistance	k^2
Wire Capacitance	1
RC Delay (unrepeated)	k^2
RC Delay (repeated)	$\sqrt(k)$
Energy per bit	$1/k^2$

However, there is an increase in complexity and associated power consumption without linear speedup. Thus, the performance acceleration from ILP has stalled.

Memory Wall. Up until the last decade, processor speed improved by 50-100% per year, while DRAM speed improved by 7%. [41] The growing disparity of processor speed and DRAM memory access latencies results in a doubling of the gap every 1-2 years. While there are more aggressive ILP extraction techniques, the memory access latencies and slow DRAM performance scaling overshadows the processor speed improvements. Processors employ a hierarchy of memories, known as caches, to achieve low-latency access to memory and effectively mask high DRAM memory access latencies, thus yielding higher application performance.

Power Wall. Up until 2004, microprocessor clock frequencies kept increasing and the diminishing size of transistors readily allowed this. However, it became extremely challenging to reduce the operating voltage, and remain within a reasonable power dissipation. Designers hit the power wall and turned to multicore processors as a result. Rather than creating new single-core processors that may not compute faster, higher performance can be gained by have many slower single-core processors working in parallel. However, power consumption is a constant constraint, even for

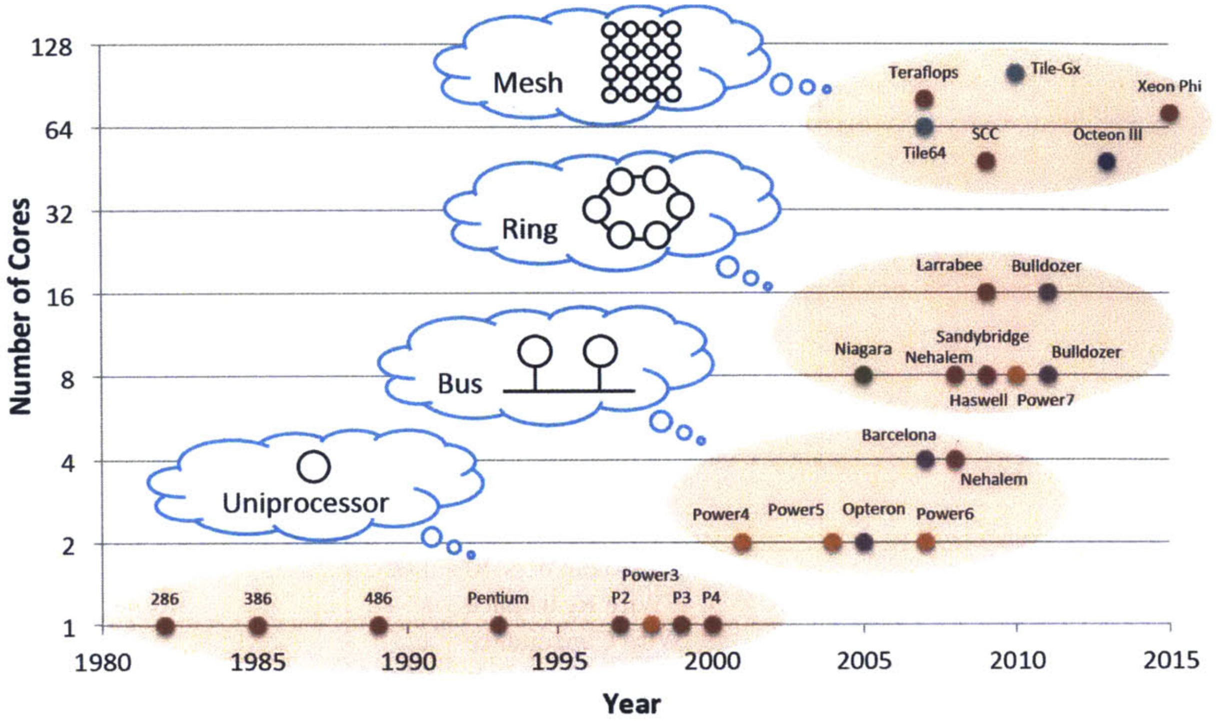


Figure 1-2: Microprocessor Core Count and Interconnection Topologies

multicore processors, and drives many circuit and system innovations.

Interconnecting the multiple processors on chip poses performance, power, and reliability challenges. As the technology node size is decreasing, the dominant delay that is emerging within multicore processors is not the gate delay, but rather wire delay. [42] Dennard articulated interconnect resistance and capacitance scaling per unit length, depicted in Table 1.1, and further identified the impact of RC delay scaling for unrepeated and repeated wires. Scaled interconnects do not scale, and provide constant RC delays since the reduction in wire capacitance is countered by an increase in wire resistance. Thus, multiple clock cycles are necessary to traverse from one edge of the chip to the other. Yet, future multicore architectures require high-bandwidth communication, scalability, modular designs, and low latency.

1.2 Scalable On-Chip Interconnects

Figure 1-2 depicts the evolution of on-chip networks for processors as the number of cores increased over the last few decades. Global wires, that cross the chip, are problematic for large multicore systems and smaller features sizes, due to the delay and power consumption growing faster than the logic gate delay. These global wires also introduce design and floor planning challenges as a result of routing congestion and long critical path delays, resulting in difficult timing closure. Traditional on-chip interconnects, such as buses and point-to-point networks, specifically rely on global wires and are unable to support high-bandwidth communication. [25]. A modular tile approach alleviates the need for global wires by utilizing *routers* to send messages/packets across the on-chip network instead of on dedicated global wires. Packet-switched network-on-chips (NoCs) enable scalable network designs for multicore processors. Such networks contain a topology of routers connected via short point-to-point links. Centralized arbitration is displaced by distributed arbitration performed at each router in the packet-switched NoC, enabling the multiplexing of multiple communication flows across the network, resulting in higher communication bandwidth.

On-chip interconnect network design comprises of trade-offs between practical constraints and performance goals. Unlike the off-chip network, on-chip networks present unique challenges where power and area are first order design constraints for new network architectures. The routers in on-chip networks consume the majority of the network power and latency cost, while in off-chip networks the network links and transmitter and receiver circuitry dominates the power consumption and latency cost. Over-provisioning the network routers may achieve higher performance, however it does not comply with the practical constraints, yielding unfeasible NoC architectures. Considerable research has been dedicated to router and network architectures to reduce latency, increase bandwidth, and consume low area and power. Buffers within the network are the main contributor to NoC area and power, yet they enable multiplexing of packets onto wires and are the main reason behind NoCs' scalable bandwidth. Although many techniques attempt to reduce the network buffer power overhead by power-gating or network buffer area by avoiding buffer writes at bypassed routers, the ideal is to achieve high-performance network communication without any network buffers. However, prior proposals advocated bufferless on-chip networks but targeted its use for low and medium workloads, as they were unable to extract low-latency high-throughput performance.

While on-chip packet-switched networks are developed to scale to many cores with ease, hardware-based cache coherence schemes are impractical for high core counts. Primarily the directory associated with managing the on-chip sharers and serving as an ordering point, is detrimental to both the area overhead as well as performance. Whether distributed or centralized directories are utilized, traversing to the remote directory, waiting for access, and being redirected to complete the request, penalizes the network latency and overall on-chip communication performance. A coherency wall [54] is observed for large core counts, leading to scalable cache coherence being imperative for future manycore processors.

1.3 Coherency Wall

As the number of cores increase in a cache coherent system, the cost associated with supporting coherence continues to grow. The rate of growth may reach a point where it is no longer feasible to maintain hardware-based coherence. The point at which this happens is referred to in [54] as the *Coherency Wall.* A cache coherent system is "scalable" if the rate of growth is (at most) on the order of the core count. Key concerns that emerge while scaling coherence are:

1. **Storage Overhead** – cost of tracking on-chip sharers
2. **Uncore (Caches+Interconnect) Scaling** – on-chip network latency and bandwidth
3. **Area and Power Consumption** – impact of coherence support on the full system cost

Directory protocols for handling cache coherence initially contained a centralized directory which serialized all requests. To keep track of the on-chip sharers of a cache block, the full-bit directory schemes allocate a directory memory proportional to the product of the total memory size and number of processors. Thus the directory size grows as $\Theta(N^2)$, where N is the number of processors. Distributing the full-bit directory across the network, improves the directory bandwidth and reduces the delay associated with serializing requests.

Scalable directory coherence protocols require the storage requirements is alleviated in these full-bit directory schemes. Limited-pointer directory [9] protocol avoids the high memory overhead of full-bit directory protocol by allowing only a constant number of simultaneously cached copies of a cache block. The size grows as $\Theta(N \log N)$ with processor core count. Since only a fixed number of pointers are allocated per entry, the storage overhead is bound but the network traffic may increase due to invalidations.

The coherent traffic sent on the interconnect requires a scalable network capable of providing high bandwidth while scaling to many cores. Read misses are independent of the number of cores as the request is forwarded to a single core within the multicore system. Write misses incur a worst case of 2N messages for N sharers representing the invalidation and acknowledgement messages when all cores are sharers of a block. Each write miss that causes N messages is preceded by N previous read misses. Thus the traffic overhead associated with a write miss is offset by the prior read misses. [17] The analysis is primarily from the coherence protocol standpoint. However, overall performance suffers as a result of contention within the network and the waiting delay at the ordering point or the directory in this case. Overall scalability suffers due to the high storage requirements.

Snoopy coherence is widely used in small multicore processors interconnected together via a shared bus. While snoopy coherence protocols broadcast each coherence transaction to all cores, they are still attractive since they allow for cache-to-cache transfers, improving performance. Snoopy coherence completely eliminates the need for the large storage overhead of directories, which becomes costly as core count increases.

The main limitations of utilizing snoopy coherence are the reliance on ordered, unscalable interconnects and the bandwidth requirement associated with broadcasts. Transitioning from ordered bus-based on-chip interconnects to packet-switched networks provides the necessary evolution to scalable on-chip networks. However, packet-switched NoCs do not inherently support ordering of requests, and hence snoopy coherence usually requires the use of ordered interconnects, such as buses and crossbars. The challenge lies in extending snoopy coherence to scalable, mesh, unordered interconnects while considering practical constraints, and pushing it towards the ultra low power and area goals.

1.4 Thesis Statement and Contributions

Snoopy coherence is a high performing, simple protocol that is easily accepted and understood by programmers and designers. It inherently provides wonderful properties of the transparency of all requests and sequentially consistent behavior in multicore systems. It is already a low cost solution to coherence, as directory coherence requires an unscalable directory to maintain coherence states,

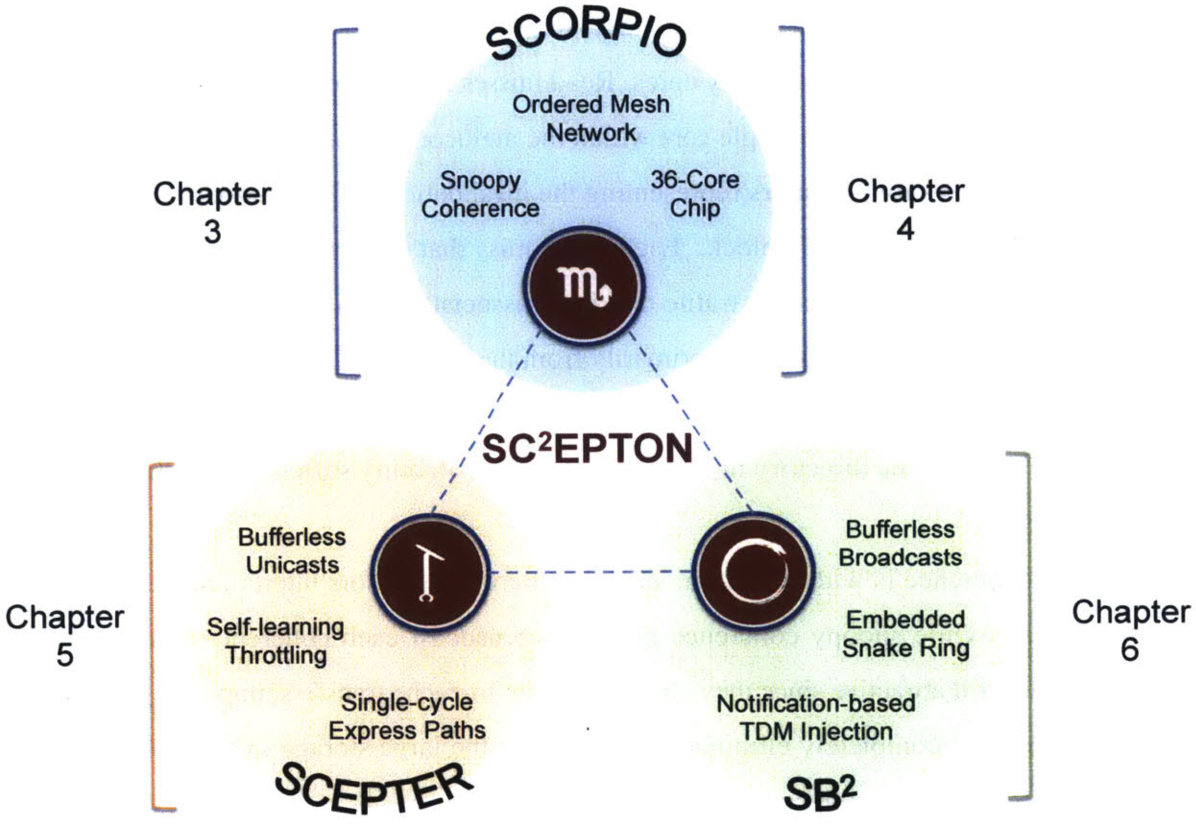


Figure 1-3: SC^{2}EPTON Overview and Thesis Contributions

however the network itself consumes a significant fraction of the tile area/power for high bandwidth communication. Of which, the input router buffers are the primary reason as they contribute to approximately 50% of the router area/power.

Figure 1-3 showcases the entire thesis, which culminates into the SC^{2}EPTON (Snoopy-Coherent, Single-Cycle Express Path, and self-Throttling Ordered Network) architecture - a low-power bufferless architecture capable of high performance communication for snoopy coherence over an ordered mesh network. SCORPIO is a novel NoC architecture that supports snoopy coherence on a mesh network with the use of common knowledge for distributed global ordering. The ordering mechanism and 36-core chip prototype development is detailed in Chapter 4. Although SCORPIO outperforms directory-based approaches, the chip prototype reveals the high cost of the router buffers even with minimal sizing for good performance. SCEPTER addresses this problem with a high performance bufferless NoC that leverages asynchronous repeated links, and maximizes opportunities to zoom along these express paths. However, SCEPTER is tailored for unicast communication, such that broadcasts are sent as multiple unicast messages. This results in increased network contention and high latencies for broadcasts. SB^2 is a bufferless broadcast embedded ring that allows flits un-contended network and ejection port access with a time-division multiplexed approach for distributed arbitration.

The major contributions of this dissertation are summarized below:

1. **High Performance and Scalable, Snoopy Coherent Mesh Network.** Snoopy coherence is supported on unordered mesh interconnects by decoupling the message delivery from the message ordering. All nodes are notified of incoming coherence request sources such that a global common knowledge is established. Subsequently, each node locally orders requests in a *consistent* manner to achieve distributed global ordering. The ordering priority rotates each time interval as all nodes maintain synchronized time intervals.

2. **36-Core SCORPIO Chip Prototype.** The chip demonstrates the ease of integration of many in-order snoopy coherent cores with a scalable network that ensures global ordering. It provides further insight into the feasibility of the approach and usefulness for the quick development of real-world multicore processors. A simple, intuitive, and scalable multicore design shows that such a high-performance cache-coherent many-core chip can be realized at low power and area overheads.

3. **Low-Power and Intelligent Bufferless On-Chip Communication with Single-Cycle Express Paths.** Achieving cache coherence with a low-cost network is very desirable. Without the overheads of the directory, snoopy coherence already places reduced storage burden within the processor. However, for performance reasons the network routers are filled with buffers. SCEPTER is a bufferless network architecture that pushes towards high performance by intelligently prioritizing, routing, and throttling flits to maximize opportunities to bypass on dynamically set, virtual express paths. It performs on-par with state-of-the-art buffered networks.

4. **Dynamic TDM-based Bufferless Broadcast Communication.** Broadcasting on top of already bandwidth-constrained bufferless networks, prompts performance concerns as snoopy coherence may not benefit as much from cache-to-cache transfers. SB^2 achieves broadcast

communication on a bufferless ring using synchronized time intervals and a time-division multiplexed network access policy. All nodes are notified of sources that require network access, where each determines the control signals for the local router.

5. **Completely Bufferless Coherent Many-Core Architecture.** The three-fold NoCs - SCORPIO, SCEPTER, SB^2 - contribute key elements to the SC^{2}EPTON completely bufferless and coherent network. This network pushes towards high performance cache coherence, while eliminating the directory storage overhead, lowering power consumption, and reducing the on-chip network traffic from broadcasts.

CHAPTER 2

BACKGROUND

An overview of the necessary background on on-chip interconnection networks , cache coherency, and memory consistency is provided as a foundation for understanding the rest of the dissertation. Please note that due to the large scope of these topics, every aspect cannot be exhaustively described and mentioned here.

2.1 Network-on-Chip

The network-on-chip (NoC) is a on-chip communication fabric that serves as a medium for messages sent from one core to another or off-chip to main memory.

Packet-switched NoCs are becoming the de-facto standard for providing scalable bandwidth at low latency for multicore systems. Each *message* sent from a processor core is divided into *packets*, and even smaller units known as *flow-control units*(flits), for distribution across the network to the destination(s). A flit is composed of one or more physical-digits, known as *phits* which are the number of bits capable of transmission on a physical link in a single cycle. As shown in Figure 2-1, flits may arrive out of order and at any time at the destination(s), depending on the routing algorithm and network traffic/contention. The NoC consists of a sea of routers interconnected via links and form the basic communication fabric of packet-switched interconnects, where each router manages the multiplexing of flits from different input links onto the desired output links. NoC designs require consideration of the topology, routing, flow control, microarchitecture, and network interface.

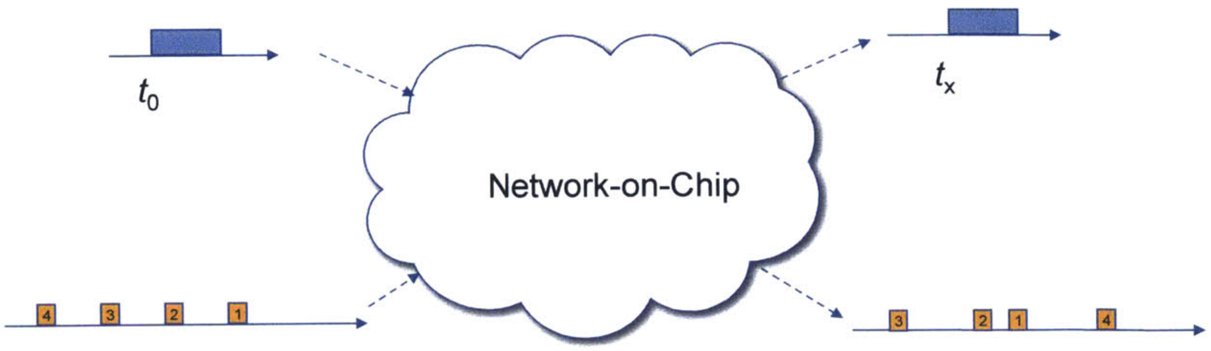


Figure 2-1: Network Communication with Packets

1. **Topology.** Physical layout and connection between routers and links.
2. **Routing.** Algorithm that determines the path a flit will take to reach its destination(s).
3. **Flow Control.** Responsible for the allocating and de-allocating of router buffers and links to different input flits.
4. **Router Microarchitecture.** Components contained within a router and the pipeline
5. **Network Interface.** Interface between the network and other components, such as the processor core.

The most universally applicable metrics of on-chip networks are latency, bandwidth, power consumption, and area usage. Latency and bandwidth can be classified as performance metrics, while power consumption and area usage are the cost factors. The *zero-load latency* is the lower bound on the average network latency as it refers to the network latency when there is no resource (router buffers and links) contention. The zero-load latency is the product of the average number of hops or routers from source to destination and the delay incurred by those routers and links. The network latency and bandwidth vary based on the topology.

Circuit-switching is an alternative to packet-switching where multiple links are preallocated for the entire message using a reservation probe to preset the links from the source to destination. It removes the need for buffers and has the advantage of low latency but is unable to provide high bandwidth. Packet-switching thus dominates as multiple traffic flows can be serviced simultaneously.

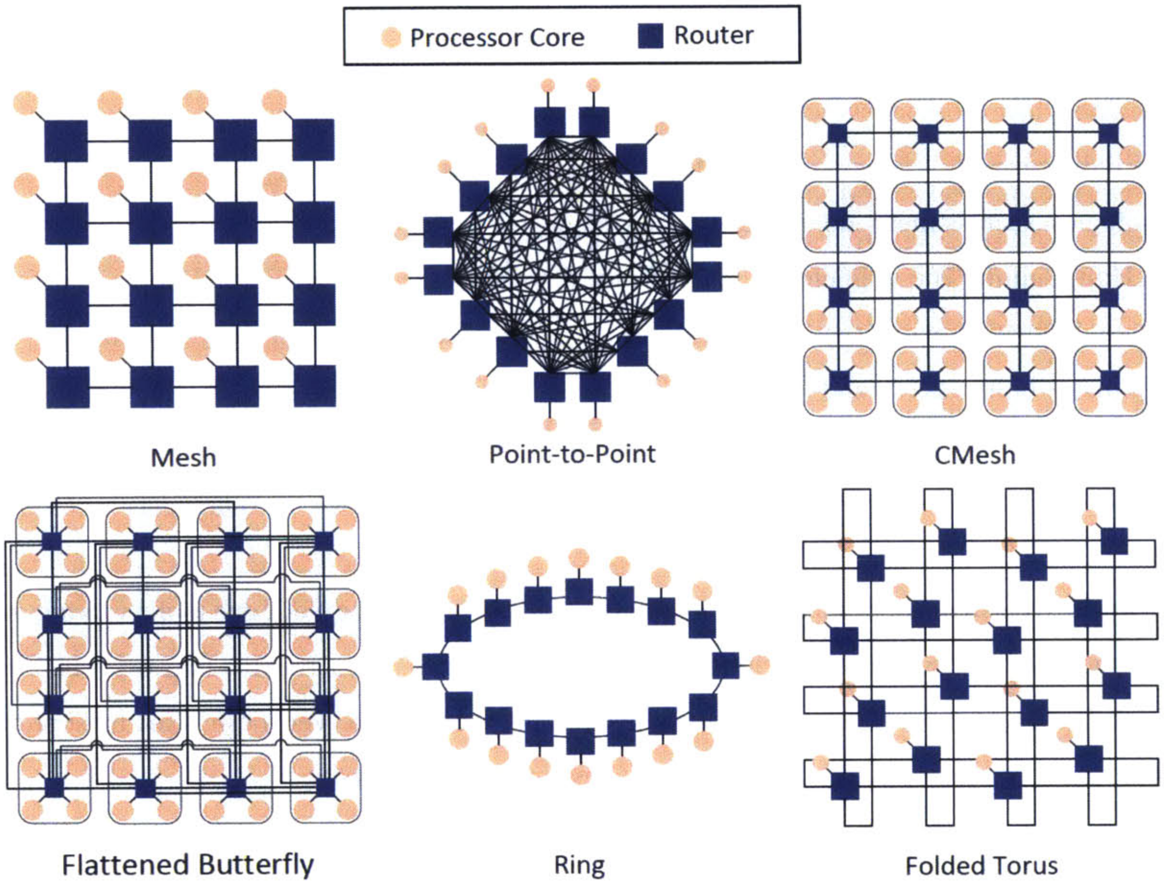


Figure 2-2: Network Topologies

2.1.1 Topology

The physical layout of the routers and the connections between them via links is referred to as the topology. The number of routers and links between nodes and the communication latency is directly affected by the chosen topology. Figure 2-2 shows a few interconnection network topologies used in packet-switched NoCs. The topologies are evaluated by the number of links at each node (degree), number of unique paths between source and destination (path diversity), average number of hops between source and destination (average hop count), maximum traffic the network can support (bisection bandwidth), and ease of layout and physical implementation. The ring interconnect is very simple and only requires simple routers and few links. The torus on the other hand is more difficult to realize, but yields a smaller average hop count and increased path diversity.

Degree. The number of links at each node, or degree, is a metric used to represent the cost of the network. A higher degree means there are more ports at each router, requiring more buffers, utilizing wider crossbar switches, and increasing overall design overhead. The ring topology has two links per node ($degree = 2$), and torus has four links per node ($degree = 4$). While some topologies, such as mesh, do not have a uniform degree for all nodes.

Hop Count. The diameter of the network defined as the largest minimal hop count for all source-destination pairs, is an indicator of the average hop count of the network. For example, for a ring that is bidirectional and has 9 nodes, the worst-case hop count is four. While a mesh would also have a worst case hop count of four for a 9-node network, a torus would reduce it to two hops. However, when interconnecting more nodes the mesh begins to surpass the ring in terms of average latency.

Path Diversity. The higher the path diversity, the more robust the NoC, due to the varying paths a flit may take in the event of failures. Traffic is also well-balanced across the network when there are multiple paths to a destination and adaptive routing is employed. The mesh is the most popular NoC topology as it is scalable, easy to layout, and offers path diversity.

Bisection Bandwidth. The bandwidth supplied by a network is a standard metric used for comparison. This is determined by splitting the N-node network in two groups of $N/2$ nodes such that the number of channels is minimal. Thus the maximum bandwidth supplied by the network is usually constrained by these bisection links. For example, a ring network with N nodes has a bisection of 2 due to the minimal bisection intersecting two channel links. In N-node mesh network the bisection bandwidth is thus $sqrt(N)$ times the link bandwidth.

2.1.2 Routing Algorithm

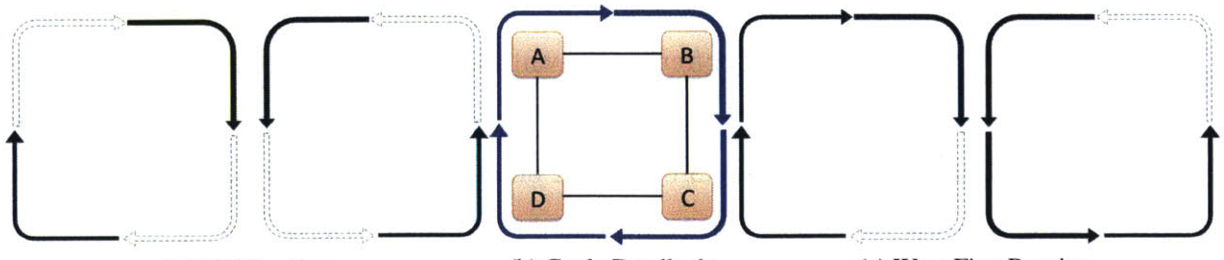
Once the topology is fixed, the routing algorithm is responsible for determining the path a flit should traverse in the network from its source to destination. While these paths could be deterministic and choose the same route from a source to a particular destination, they could also adapt to the network conditions and choose alternate paths while hopping through the network. Three main classifications of routing algorithms are: deterministic, adaptive, and oblivious.

Deterministic. Deterministic routing schemes are straightforward and simple as there is a set route irrespective of the network traffic contention. Dimension-ordered routing is a popular deterministic routing algorithm due its simplicity. A flit traverses along one dimension, X or Y, completely, prior to traversing the other dimension to reach the desired destination node. For example, a flit injected at node (x=1,y=1) would like to be sent to node (x=2,y=4). With XY routing, the flit would move one hop in the X direction followed by three hops in the Y direction. Deadlock freedom property is maintained if the permitted routes are free of cycles. [34] Permitted turns are shown in Figure 2-3 as it illustrates that a routing cycle is impossible.

Adaptive. Packets traverse different paths between the same source-destination pair when using adaptive and oblivious routing algorithms. Adaptive routing uses the state of the network when determining the selected path. Minimally-adaptive routing limits the selected paths to those with the shortest distance from the current node to the destination. Non-minimally adaptive routing considers other paths, even those that result in the packet diverting from the minimal route. To ensure deadlock freedom, adaptive routes consist of certain restrictions that prevent routing cycles.

Oblivious. In contrast to adaptive routing, oblivious routing does not utilize the network state when determining the flit route. Valiant [79], an oblivious routing protocol, randomly selects an intermediate node to route to, from which it routes to the destination. This spreads the traffic such that it is uniformly distributed across the network. A higher latency cost is paid to achieve the load balancing.

Deterministic and oblivious routing algorithms are low overhead solutions for route computation. On the other hand, adaptive routing implementations are non-trivial as deadlock freedom must be ensured and large routing tables may be necessary.



(a) XY Routing (b) Cycle Deadlock (c) West-First Routing

Figure 2-3: Avoid Deadlock with Permitted Turns that Prevent Cycles

2.1.3 Flow Control

Flow control determines the allocation of network resources such as buffers and channels to flits. A good flow control mechanism reduces the latency for low loads without high overhead. Throughput is also sensitive to the flow control mechanisms as effective sharing of network resources ensures higher bandwidth.

In *store-and-forward* flow control, the entire packet, composed of multiple flits, is entirely buffered at a node prior to being sent to the next node. It incurs high network latencies; making it unsuitable for on-chip networks. *Virtual cut-through* flow control tackles this problem by allowing flits to be sent prior to receiving all the flits, within the packet, at each node. However, the flow control is performed at the packet-level, where the packets only move forward if there is enough buffer space for the whole packet.

Wormhole flow control is a flit-level flow control scheme that allows buffer allocation and channel traversal to be performed flit by flit. Hence, flits are able to move along to the next router even before the entire packet is received at the current router. Bandwidth and buffer space is managed on a fine grained scale of flits, much smaller than packets. Since buffers are allocated on a flit-basis, the required per-router buffer space is much lower than packet-level flow control. However, links are held until the entire packet reaches the destination. Thus it is susceptible to head-of-the-line blocking, where a flit behind a blocked packet is unable to use the idle links. *Virtual-Channel (VC)* flow control alleviates this problem by using virtual channels. A virtual channel is a separate flit queue within the router where multiple VCs arbitrate for access to the outgoing physical links. Thus if a packet is blocked in one VC, another VC's flit is still able to access the links and be sent through to the next router.

Since the network should not allow the dropping or loss of packets, the flow control must ensure an arriving packet has the necessary buffer space available. Two common ways of achieving this is on-off or credit-based signaling. In *on-off signaling*, downstream routers inform the current router of the buffer status by toggling a bit. If the bit is asserted, the number of free buffers is above a threshold value, otherwise it is below that threshold. In *credit-based signaling*, a count is maintained at each router, indicating the number of free buffers at the downstream router in a certain direction. Each time a flit is sent, the count is decremented. When the downstream router

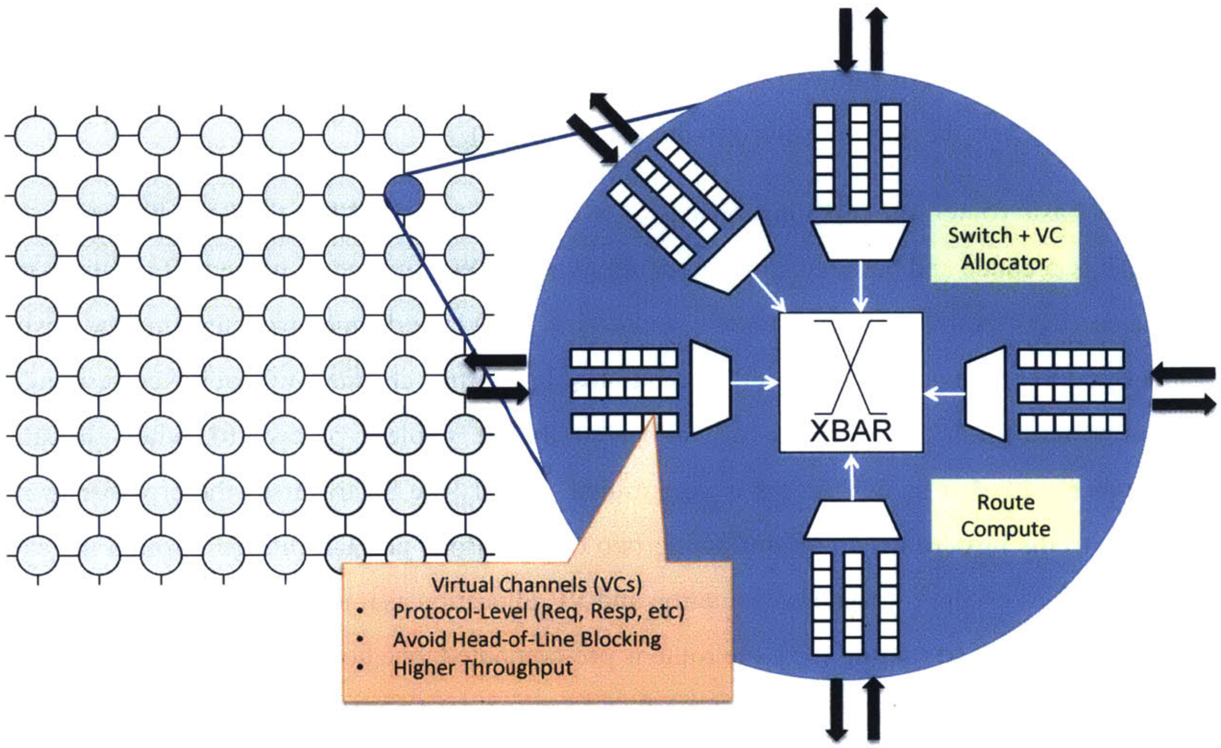


Figure 2-4: Router Microarchitecture

buffer space becomes available, it sends a credit back to the upstream router which then increments the credit count.

Buffer turnaround time is the minimum delay between when a buffer is occupied, and flit leaves, to the arrival of the next flit to be buffered. The delay consists of (1) indication of a free buffer, via on-off or credit signaling (credit/on-off propagation delay), (2) update the credit count at upstream router (credit/on-off pipeline delay), (3) send the flit because buffer space is available at downstream router (flit pipeline delay), (4) leave the router to be sent to downstream router (flit propagation delay). Thus, longer router pipelines and interconnecting wires lead to poorer buffer utilization keeping buffers unused longer.

2.1.4 Router Microarchitecture

To meet the latency and throughput goals, the router must be carefully designed as it directly affects the network latency, maximum frequency of the network, and network area/power. The microarchitecture of the virtual-channel router is shown in Figure 2-4. Each input port contains multiple VCs, all able to arbitrate for outgoing links. In the mesh network shown, the router has five input and output ports corresponding to the neighboring directions, east (E), west (W), north (N), south (S), and the port connected to the local node (L).

For a basic router, the pipeline consists of five logical stages. When the head flit arrives at a router, it is decoded and buffered, known as buffer write (BW), and occurs in the first pipeline stage. In the second pipeline stage, the route computation (RC) is performed to determine the desired output port(s) of the packet. The flit then arbitrates for a virtual channel (VC) at the corresponding output port. If the VC was successfully allocated, the flit is able to proceed to switch allocation (SA) where crossbar switch access is determined. Finally the flit traverses the crossbar switch and link to the next router in the following two cycles. Body and tail flits go through the same pipeline stages except for route computation and VC allocation as it is predetermined for the head flit. Using lookahead routing [33], the route is pre-computed, one hop in advance, which reduces the pipeline to 4 stages. After the BW stage, flits are able to perform VC allocation, followed by the SA, ST, and LT. To further reduce the pipeline depth, the VA and SA can be performed in parallel by speculatively performing the switch allocation after the BW stage. The pipeline is shown in Figure 2-5. The flit arbitrates for the switch access while also determining if there is a free VC available at the downstream router. If a VC could not be allocated, the speculative SA fails and the process repeats until success ensues. Fair allocation is essential to ensure certain flits are not starved.

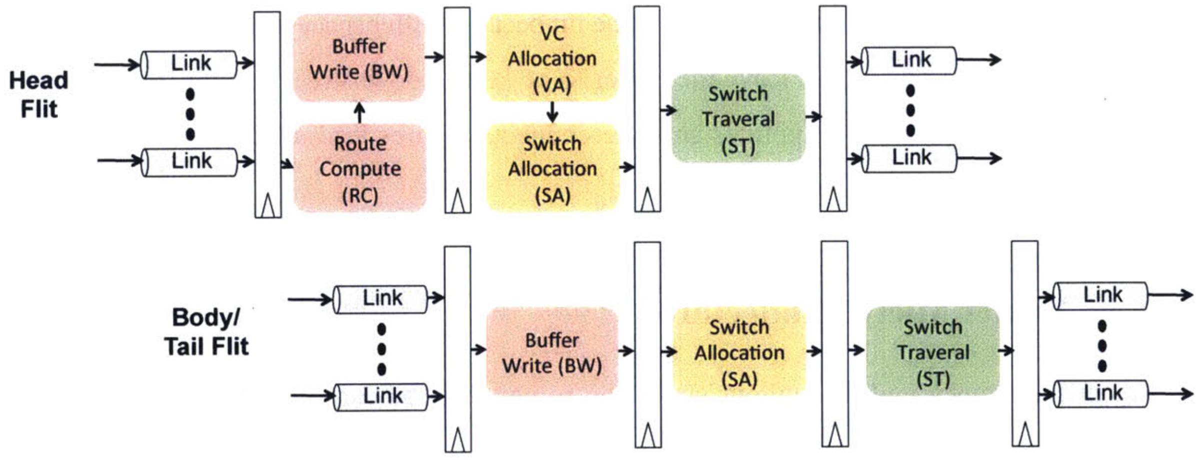


Figure 2-5: State-of-the-Art Virtual Channel Router Pipeline

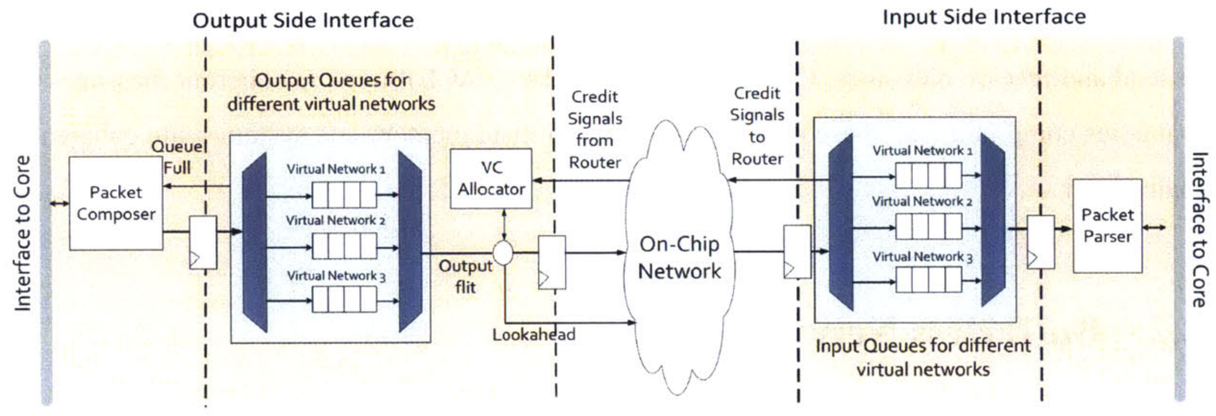


Figure 2-6: Network Interface Allowing Communication between Cores and the NoC

2.1.5 Network Interface

To interface the network with the outside world, namely the cpu/caches, directory, and memory controllers, a network interface is required. The network interface (NIC) typically connects directly between the router and private L1 or L2 cache in a shared memory system. The multicore system communicates via messages, which are converted by the network interface, shown in Figure 2-6, into packets to be sent across the NoC. Each message from the processor core is encapsulated into packets with regard to the network specific details of channel-width and routing algorithm. For instance, a data response message, consisting of the entire 32-byte cache line, is sent from the on-chip sharer to the requester. If the packet size is 16 bytes, the message must be divided into two packets. Packets are divided into flits, and sent into the network, one by one. Each packet consists of a head flit that holds the destination address, multiple body flits, and a tail flit. Assuming a flit size of 64 bits, the 16-byte packet is divided into 2 flits, ignoring the encapsulation of network routing information. Thus, three flits are necessary to incorporate part of the cache line and the destination and flow control information. The flits remain in the NIC's virtual channel buffers awaiting access to the network. Using credit signals, the router indicates to the NIC the status of the buffer space in the NIC input port, such that a flit is sent to the router if space is available.

Once the destination receives all the flits, the packet parser combines the flits into a packet and parses through the contents. The interface to the core is responsible for converting packets into messages, and communicating the message to the core. Many processor cores contain standard interfaces, such as OCP [2] and ARM AMBA [1], which utilize standard communication protocols to send and receive messages. OCP and ARM AMBA 4 (ACE) supports coherent messages such that cores compliant with these interfaces are able to share memory and system-wide coherency is maintained.

2.2 Bufferless Networks

On-chip networks are becoming prominent in multicore systems, however the consideration of area/power overheads is affecting the performance potential. Designs attempt to reduce the power of interconnects by low-swing signaling, power gating, and DVFS. However, upon closer inspection of the router area and power consumption, it is evident that the input buffers consume a significant portion of the total tile area/power.

Bufferless NoCs eliminate the need for buffers within the network. Flits contending for the same ports are either deflected towards other ports, or dropped awaiting retransmission by an upper layer protocol. As a result, bufferless NoCs have in the past traded off performance for low area and power overheads. The baseline bufferless NoC router is shown in Figure 2-7. The two-stage router pipeline performs route computation and priority output port allocation (referred to as switch allocation throughout this paper) in the first pipeline stage, followed by switch and link traversal in

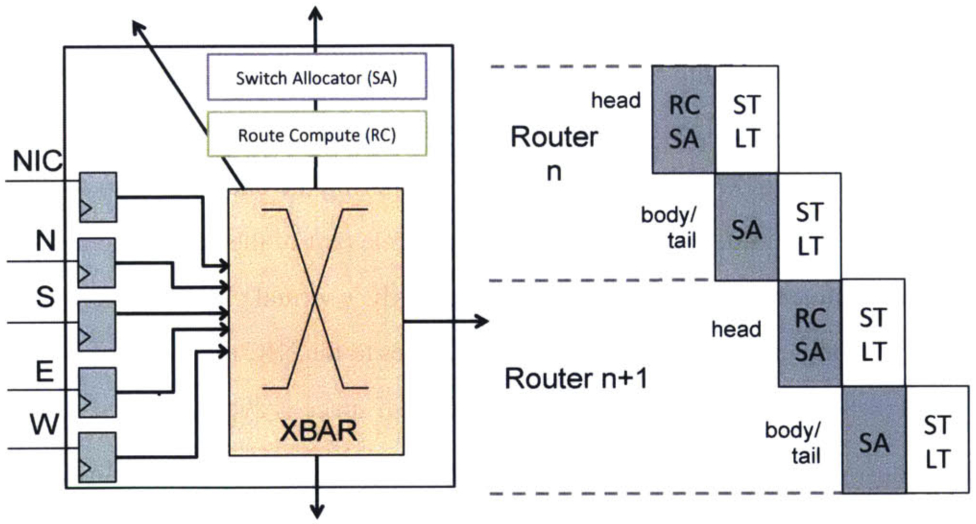


Figure 2-7: Bufferless Router and Pipeline

the second pipeline stage. Flits are temporarily held in pipeline registers within each router and between each router pipeline stage, until an output port allocation is performed.

2.2.1 Deflection Routing

Bufferless networks and deflection routing have been developed and used in the Internet and optical networks, and is more commonly known as hot-potato routing (see Chapter 5 for more background on bufferless networks in other domains). Recently, bufferless NoCs have become appealing due to tight on-chip power constraints. Similar to hot-potato routing, deflection routing in NoCs requires that a flit does not reside in a router, thereby it continually moves within the network. Since flits cannot be buffered, all the flits arbitrating at the router, in this cycle, need to be assigned an output port and leave the router. If the assigned output port does not lead closer to the destination, it is known as a *deflection*. The baseline bufferless NoC is based on BLESS [65], a deflection-based bufferless NoC where the route is computed using deterministic XY or YX routing. The deflections intrinsically achieve adaptive routing because flits can be routed along multiple paths to the destination.

2.2.2 Allocation

BLESS uses an *Oldest-first* prioritization rule to arbitrate among the incoming flits. A global consistent order is maintained by using this age-based approach. However, this approach requires a timestamp/age to be propagated with every packet, and a priority sorting that incurs a long arbitration critical path. The timestamp must be wide enough to account for the largest in-flight window, i.e thousands of cycles for a 256 node NoC, if not more, as it depends on the number of requests in the network, timestamp rollover/reuse, and traffic pattern. Flits are processed in order of priority, where the highest priority flit obtains the desired output port and the other flits choose among the unallocated ports, where productive ports are prioritized over non-productive ones. Using age-based prioritization, livelock freedom is ensured as the globally oldest flit will always win its desired output port allocation at each hop.

2.2.3 Injection

Within a bufferless router the incoming flits must be assigned to an output port. Thus, flits are constantly moving around the network until the destination is reached. It provides deadlock freedom guarantees but does not ensure that a local node, i.e. network interface of core or cache, is always able to inject a flit into the network when necessary. For a flit to be injected, there must be an idle input slot on one of the North, South, East or West links. This guarantees that the local flit will obtain an output port during switch allocation. This prompts starvation and fairness concerns as other cores' traffic may monopolize the network bandwidth and prevent certain cores from injecting.

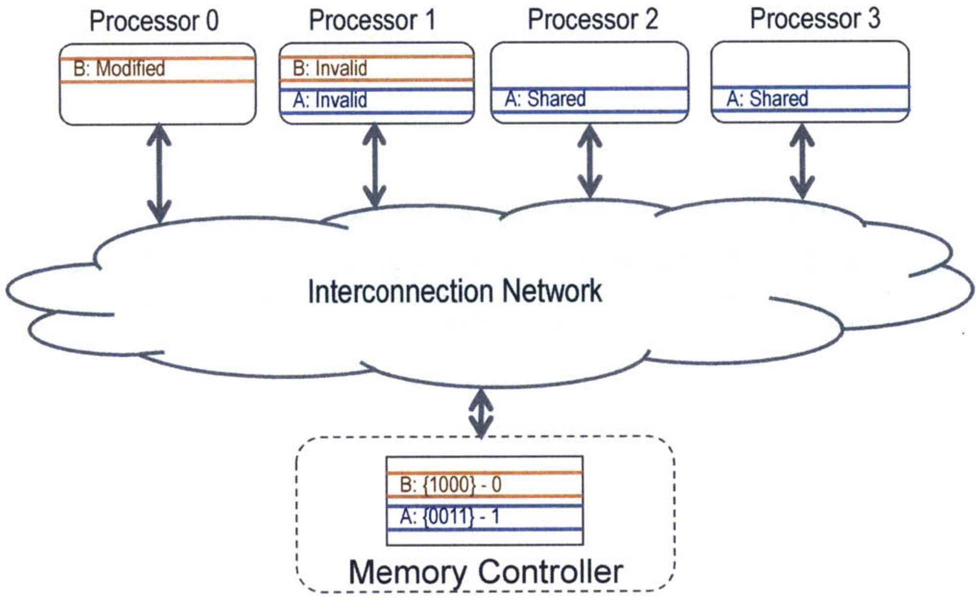


Figure 2-8: The Problem of Incoherence in Shared Memory Systems

2.3 Cache Coherent Interconnects

In a shared memory system the data responses and integrity should be guaranteed and uncorrupted. However, this is quite challenging when multiple processors are all accessing and computing on shared memory. To illustrate this incoherence problem, consider the 4-core system in Figure 2-8, but for now ignore the cache block state information shaded. Processor 3 (P3) and processor 2 (P2) both load cache block A into their private caches. Processor 1 (P4) retrieves and stores cache block

B, and processor 1 (P1) is idle. Assume P2 performs a write to the cache block A, it will update the value in its private cache. Since P3 is also caching block A, it retrieves the data from its private cache, computes on it, and writes it back to its local cache block A. Now, we have two processors that are unaware of the modifications made by other caches. P1 wishes to retrieve cache block A into its local cache to compute on it. *What is the correct value to send to P1 - the result of P2's write or P3's write?*

The solution is to maintain the coherence single-writer, multiple-reader invariant; that is, there is only a single writer at a time, while there may be multiple readers of a cache block. The result of a read to a memory location must return the value from the last write to that memory location. Otherwise, cores use the stale data and incorrectly compute with that data. In the example provided, P3 computes with stale data and when that data is written to A's location, it will affect the entire program and yield incorrect results.

Cache coherence protocols are used to assure the data integrity in shared memory multicores. Each cache block stored in a local processor's caches are assigned a state representing the permissions granted to this processor. Thus, P3 should not be granted permission to modify the value of A prior to being informed of the previous write by P2. The valid states given to cache blocks vary based on the coherence protocol used. A simple one is the MSI protocol, where the three states represents if the cache block is invalid, shared, or modified within the private cache.

- *Modified (M):* The cache block is within the private cache, and is granted both read and write access. This cache is responsible for responding to any requests pertaining to this cache block.
- *Shared (S):* The cache block is within this private cache, and is granted read access only.
- *Invalid (I):* The cache block in the private cache is not valid. The contents are not usable and if needed, the cache block should be retrieved first.

Transitioning from one coherence state to another, for instance state *Shared* to state *Modified* upon a write operation, begins with the core fetching and decoding the respective load or store instruction. The coherence transaction is sent out of the core if the local cache does not have the correct permission to service the type of request. For instance, a load instruction is generated by the core, however the cache block has an *Invalid* state and the data must be obtained from main memory. A request is generated and sent to main memory. Prior to the arrival of the data response, and after the generation of the request, the cache block is in a "transient" state. It differs from the three "stable" states (MSI) mentioned as it does not reside in either of these states while the request is being serviced. Upon reception of a data response, the cache block state shifts from *Invalid* to *Shared*. The coherence transaction is completed once the data is appropriately moved and the state is updated.

When a processor performs a write operation to a cache block, in the *S* state, the state needs to transition to *M*. To ensure we do not encounter the incoherency problem, other caches with the cached copy of the data needs to be invalidated prior to performing the data write. Invalidation messages are sent to other caches informing them to change their cache block states from *S* to *I*. Multiple processors can read block copies from main memory safely until one processor updates its copy. At this time, all cache copies are invalidated and the memory is updated to remain consistent.

The coherence protocols are improved for performance by adding additional states. The *Exclusive (E)* state indicates that this is the only cached copy of the block within the multicore system. Thus, writes can be implicitly performed without invalidations. The *Owned (O)* state is granted to a cache that is responsible for responding to remote coherence requests to this cache block. Thus, eliminating the need to obtain the cache block from main memory when it is serviced by another cache on-chip. Various common coherence protocols are formed by combining the "stable" states, e.g. MSI, MESI, MOSI, and MOESI. Table 2.1 lists a few common transactions as a result of certain coherent actions observed or initiated.

Table 2.1: Common Coherence Transactions

GetShared	(GetS)	Obtain block in Shared (read-only) state
GetExclusive	(GetX)	Obtain block in Exclusive (read-write) state
PutClean	(PutC)	Evict unmodified cache block
PutDirty	(PutD)	Evict modified/dirty block

There are two basic protocol types: (1) Write-invalidate, and (2) Write-update. The write-invalidate process was briefly mentioned earlier. Whenever a write is performed, all other locally cached blocks, pertaining to the same memory address, are invalidated. Write-update protocols do not utilize invalidations, rather the updated data block is broadcast to all caches with a locally cached block. Throughout this dissertation, write-invalidate coherence protocols can be assumed, unless mentioned otherwise.

Two main classifications of cache coherence protocols are snoopy protocols and directory protocols. Each is developed with basic assumptions of the underlying architecture. Snoopy protocols assume all the coherence controllers are able to *snoop* or see the requests from other cores. They tend to be fast if enough bandwidth is available for the broadcast of all transactions. The drawback is that it is not scalable as broadcasts cannot be sustained for large multicore systems. Directory protocols assume a directory is present to maintain the coherence state of each cache block. These protocols tend to have longer latencies as each request has 3 phases: request, forward, respond, but consumes less bandwidth since point-to-point messages are used rather than broadcasts.

2.3.1 Snoop-based Protocols

Snoop-based coherence protocols are popular in multicore systems due to its simplicity and low overhead. All coherence controllers *snoop* transactions generated by other cores and memory controllers. Thus, every update to a cache block is visible to all processors. Each processor then behaves appropriately based on the transactions observed. For instance, upon observation of a write request, the processors check if the matching cache block is locally cached. If so, the cache block is invalidated in the case of write-invalidate protocols. For write-update protocols, the updated value is obtained from the write transaction observed, and each locally cached copy is replaced with the new value.

Blocks are required to arrive in order, such that the coherence is appropriately enforced. The bus and tree are natural networks for snoopy coherence as transactions are ordered and broadcast to all. Additionally, the ordering eases memory consistency models, especially sequential consistency where total ordering of memory transactions is desired. Recent work on coherence protocols have pushed towards having the network and coherence integrate tighter to optimize transactions.

2.3.2 Directory-based Protocols

A directory-based cache controller issues a transaction that is sent to the directory. The directory looks up the state of the cache block and subsequently the request is forwarded to the *owner* of the cache block. The owner is possibly a processor core on-chip or the main memory. Since a sharer list is maintained for each block, requests do no need to be broadcast to all cores. Similar to snoop-based protocols, the memory accesses to the same location need to be ordered. The directory serves as an ordering point as requests are serialized and processed.

Full-map directories store enough states for each block in global memory such that every cache is able to locally cache a copy of any block of data. Each directory entry for a cache block contains N bits, where N is the number of processors in the system. In this bit vector, an asserted bit represents the presence of a cached copy of the block in the corresponding processor. An additional bit specifies if a processor on-chip has write access to the cache block. Full-map directories are no scalable since the space required for the bit vectors in all entries is immense. A limited directory solves this by using a fixed directory size. Rather than bit vectors, the limited directory uses a few pointers to represent the sharers of a cache block. Thus, only a limited number of caches is able to locally cache the same block simultaneously. If a new load request arrives at the directory but all pointers are accounted for, one of the caches will need to be evicted in order to assign a new pointer in its place and complete the load request.

Further optimization of directory coherence includes tracking data sharers in a coarse granularity. [16] Multiple cores are within a region where a region bit vector is stored in each directory entry. Thus, coherence requests are broadcast to all cores within a region if the presence bit is asserted, indicating at least one sharer is within that region. The AMD Hypertransport [23] commercial coherence protocol relies on the broadcast of requests as the directory only indicates if the sharer is on-chip or off-chip. These hybrid coherence protocols trades network bandwidth for storage space.

2.3.3 Protocol-Level Deadlock Avoidance

Assuming all requests and responses utilize the same set of VCs, there is a potential for deadlock. If a request was initiated by the L2 cache and sent into the network. The response is sent by another node or the directory. The network is full of requests dependent on the outcome of a previous request, which in turn is awaiting the response from the network to complete the coherent transaction. However, the response is trapped behind these requests within the network and is unable to make forward progress.

To resolve this problem, virtual networks (vnets) are used. Each type of coherent transaction in the network is placed into exclusive message classes. Each message class maps to a virtual network in the router. Virtual networks are essentially a grouping of VCs dedicated to transactions pertaining to a message class. For instance, a simple snoopy protocol consists of only requests and responses, and requires two message classes to achieve protocol-level deadlock avoidance. Although requests and responses are stored in separate set of VCs, they still all arbitrate for access to the same physical links.

2.3.4 Memory Consistency

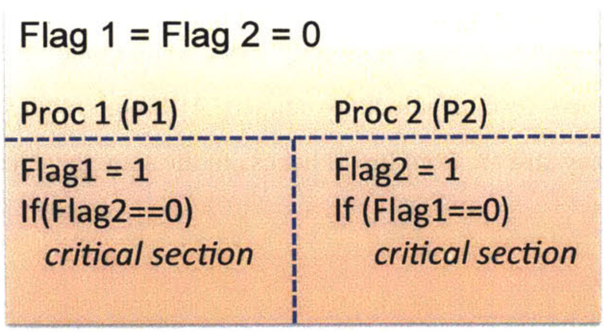
Single-core processors provide a simple and intuitive view of the memory to the programmer as instructions appear to execute in program order, even if out-of-order cores are utilized. However, memory consistency in multicore processors is not as straightforward and incurs additional hardware challenges. The memory consistency model of a multicore processor provides a specification of the memory system to the programmer. It bridges the gap between the expected and actual behavior of the system. Intuitively, a read should return the value of the "last" write to the same memory location. The last write is simply defined by the program order, order of memory accesses in the program, for single-core processors. For multicore processors, the "last" write is not as obvious as it could have originated from another processor.

The most intuitive memory consistency model for multicore processors is *sequential consistency*, defined in 1979 by Leslie Lamport as follows. [56]

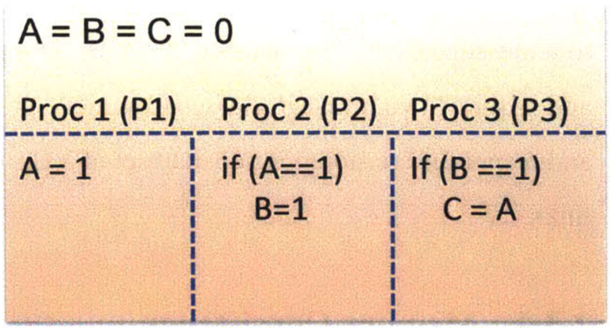
Definition [A multicore system is *sequentially consistent* if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

The definition includes two aspects of sequential consistency: (1) program order must be maintained for individual processors, and (2) a global order is maintained of operations of all processors. In other words, the system is sequentially consistent if the operations of all processors appear to execute in a sequential order, and the program order is maintained for each individual processor. Figure 2-9 shows two examples of sequential consistency.

Figure 2-9a is a simple implementation of Dekker's algorithm where two processors attempt to enter the *critical section* at the same time, and only one is allowed to enter. Two processors (P1 and P2) and two flag variables (Flag1 and Flag2) are used, with the variables initialized to zero. When P1



(a)



(b)

Figure 2-9: Sequential Consistency Examples

attempts to enter the critical section, it firsts updates Flag1 value to 1, and checks the value of Flag2. If Flag2's value is 0, P2 has not tried to enter the critical section already, and P1 is able to enter. When P2 tries to enter the critical section, it will see that Flag1 is 1, meaning P1 is in the critical section. Sequential consistency ensures this behavior by requiring program order be maintained for both P1 and P2's instructions, thus precluding the possibility of both processors reading the flag value of 0 and entering the critical section. Sequential consistency allows interleaving between instructions from different processors, so long a total global order is maintained. Thus, if both Flag1 and Flag2 are assigned value of 1 and the conditional is checked, neither P1 or P2 would be able to enter the critical section. Even with this strict consistency, the program needs to be carefully designed to achieve mutual exclusion. Figure 2-9b shows three processors accessing the same shared memory locations, A, B and C, initialized to 0. Processor P2 reads the value of A and if it was set was P1 previously, it executes the write to memory location B. The effect of P1's write to A must be seen by all cores at the same time, as sequential consistency ensures atomicity.

In a coherent system, sequential consistency is easily achieved on a shared bus, as in-order execution of memory access is guaranteed. However, this becomes difficult to support in unordered interconnects, such as mesh networks. Relaxed consistency models [8] have been proposed such that reads and writes can complete out of order. Special synchronization operations are required to enforce ordering and require programmer expertise to achieve correct and high performing applications. Cache coherence guarantees all processors consistently view memory accesses to a specific location but not across all memory locations. A coherent system may not be sequentially consistent.

2.3.5 Synchronization

Shifting from single-core processors to multicores requires prudence when writing functionally correct applications. Hardware synchronization primitives assist with mutually exclusive access to memory locations. Locks are a type of synchronization primitive that ensures only one processor gains access to a critical section at a time. The range of lock primitives is immense, which include test and set lock (TAS) and load link store conditional (LL/SC). The TAS lock was the main synchronization instruction provided in early multicore architectures. Each processor requesting the lock, will store "true" in the memory word (or byte), and the previous value is returned. A returned value of "true" indicates another processor obtained the lock, is currently in the critical section and will set to memory word to "false" when it exits. It is evident that the underlying architecture impacts the performance of the TAS lock as the number of requesters for the lock increases. Each write to the memory location incurs a flood of invalidations to other caches which have locally obtained the cache block to also perform a write. The load linked store conditional mechanism eliminates the need for excessive invalidations. LL/SC splits the process of obtaining the lock into two phases: (1) load the value in the memory location, and (2) store a value to memory location on the condition that the location has not been modified since the value was loaded. If another processor updated the memory location, the store will fail and the LL/SC is restarted. Each processor is able to locally store a copy of the cache block. Upon viewing a store from another processor the loaded cache block is invalidated. Thus, invalidations are only generated by successful stores.

CHAPTER 3

SCORPIO: GLOBALLY ORDERED MESH NETWORK ARCHITECTURE

Shared memory, a dominant communication paradigm in mainstream multicore processors today, achieves inter-core communication using simple loads and stores to a shared address space, but requires mechanisms for ensuring cache coherence. Over the past few decades, research in cache coherence has led to solutions in the form of either snoopy or directory-based variants. However, a critical concern is whether hardware-based coherence will scale with the increasing core counts of chip multiprocessors [47,55]. Existing coherence schemes can provide correct functionality for up to hundreds of cores, but area, power, and bandwidth overheads affect their practicality. Two of the three scalability concerns are (1) directory storage overhead, and (2) uncore (caches+interconnect) scaling.

For scalable directory-based coherence, the directory storage overhead must be kept minimal while maintaining accurate sharer information. Full bit-vector directories encode the set of sharers of a specific address. For a few tens of cores it is very efficient, but requires storage that linearly grows with the number of cores; limiting its use for larger systems. Alternatives, such as coarse-grain sharer bit-vectors and limited pointer schemes contain inaccurate sharing information, essentially trading performance for scalability. Research in scalable directory coherence attempts to tackle the storage overhead while maintaining accurate sharer information, but at the cost of increased directory evictions and corresponding network traffic as a result of the invalidations.

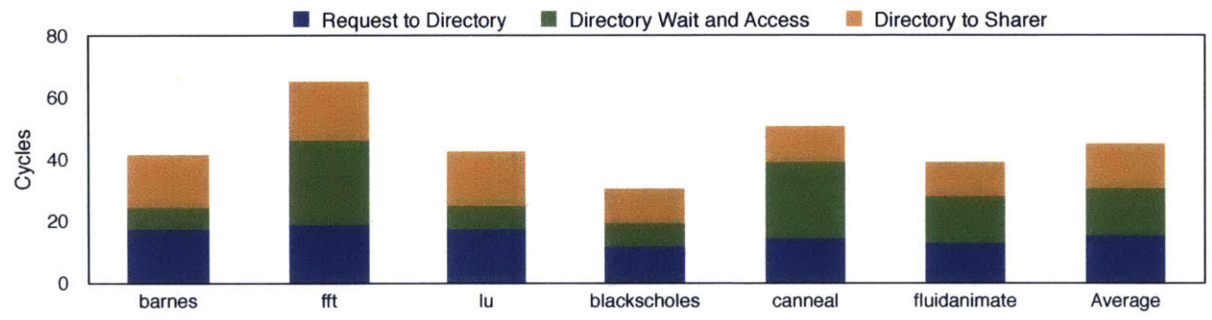


Figure 3-1: Indirection and Serialization Latency of Directory-Based Coherence

3.1 Motivation

Traditionally, global message ordering on interconnects relies on a centralized ordering point, which imposes greater *indirection*, network latency of a message from the source node to ordering point, and *serialization latency*, latency of a message waiting at the ordering point before it is ordered and forwarded to other nodes. Dependence on a centralized ordering point prevents architects from providing global message ordering guarantees on scalable but unordered networks.

Distributed directory-based coherence alleviates the latency cost by separating the directory across the chip, thereby splitting the requests among the directories by address space. A reduced serialization latency ensues, as well as indirection latency if the directory node is close to the requester. However, even with distributed directories, the latency cost is significant. Figure 3-1 shows the indirection and serialization latency for SPLASH-2 and PARSEC benchmarks on a modeled 36-node system, with distributed full-map directory-based coherence in the GEMS [59] simulator. Across all applications, the average latency cost of 45 cycles prohibits efficient communication.

Snoopy coherence exhibits higher performance by avoiding the indirection latency cost through direct cache-to-cache data transfers. However, the performance doesn't scale with the number of nodes as ordered, unscalable networks are required for correctness. Snoopy compatible interconnects comprise buses or crossbars (with arbiters to order requests), or bufferless rings (which guarantee in-order delivery to all cores from an ordering point). Existing on-chip ordered interconnects scale poorly; The Achilles heel of buses lie in limited bandwidth, while that of rings is delay, and for crossbars, it is area. Higher-dimension NoCs such as meshes provide scalable bandwidth and is the subject of a plethora of research on low-power and low-latency routers, including several chip

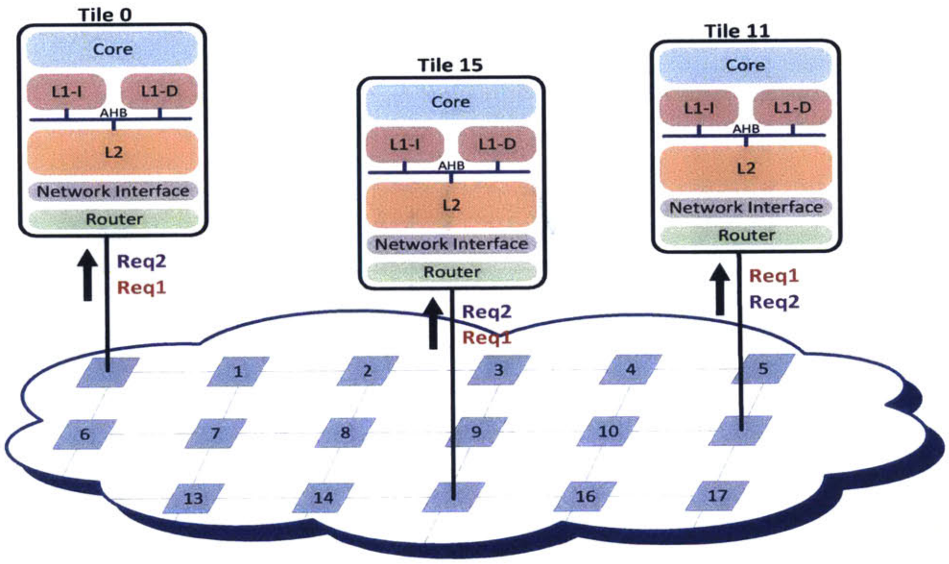


Figure 3-2: Requests Delivered to Nodes over Unordered Mesh Network

prototypes [36, 44, 68, 75]. However, meshes are unordered and cannot natively support snoopy protocols. Figure 3-2 shows an unordered mesh network tasked with delivering *Request 1* and *Request 2* to all the tiles. Flits from these requests contend for network resources within each router. It isn't guaranteed that each flit will be seen in the same order across all the tiles. Some tiles view *Request 1* prior to *Request 2* and vice versa. The ideal is an unordered scalable network with ordering mechanisms that ensures sequential consistency and snoopy coherence is maintained, while isolating the core from the details and providing high performance.

3.2 Related Work

Various proposals, such as Token Coherence (TokenB), Uncorq, Time-stamp snooping (TS), and INSO extend snoopy coherence to unordered interconnects. TokenB [58] performs the ordering at the protocol level, with tokens requested by a core wanting access to a cache line. A read request can only proceed if at least one token is obtained for the cache line, while for a write request, all tokens need to be obtained. TokenB assigns T tokens to each block of shared memory during system initialization (where T is at least equal to the number of processors). Each cache line requires an additional $2 + \log T$ bits. Although each token is small, the total area overhead scales linearly with the number of cache lines. Failure to obtain necessary token(s), results in repeated tries and potentially continuous collisions with other requests, prompting livelock concerns and degrading performance.

Uncorq [73] broadcasts a snoop request to all cores followed by a response message on a logical ring network to collect the responses from all cores. This enforces a serialization of requests to the same cache line, but does not enforce sequential consistency or global ordering of all requests. Although read requests do not wait for the response messages to return, the write requests have to wait, with the waiting delay scaling linearly with core count, like physical rings.

TS [57] assigns logical time-stamps to requests and performs the reordering at the destination. Each request is tagged with an ordering time (OT), and each node maintains a guaranteed time (GT). When a node has received all packets with a particular OT, it increments the GT. TS requires a large number of buffers at the destinations to store all packets with a particular OT, prior to processing time. The required buffer count linearly scales with the number of cores and maximum outstanding requests per core. For a 36-core system with 2 outstanding requests per core, there will be 72 buffers at each node, which is impractical and will grow significantly with core count and more aggressive cores.

INSO [12] tags all requests with distinct numbers (snoop orders) that are unique to the originating node which assigns them. All nodes process requests in ascending order of the snoop orders and expect to process a request from each node. If a node does not inject a request, it is has to periodically expire the snoop orders unique to itself. While a small expiration window is necessary for good performance, the increased number of expiry messages consume network power and bandwidth. Experiments with INSO show the ratio of expiry messages to regular messages is 25 for a time window of 20 cycles. At the destination, unused snoop orders still need to be processed leading to worsening of ordering latency.

Swizzle [71] is a self-arbitrating high-radix crossbar which embeds arbitration within the crossbar to achieve single cycle arbitration. Prior crossbars require high speedup (crossbar frequency at multiple times core frequency) to boost bandwidth in the face of poor arbiter matching, leading to high power overhead. Area remains a problem though, with the 64-by-32 Swizzle crossbar taking up 6.65mm^2 in 32nm process [71]. Swizzle acknowledged scalability issues and proposed stopping at 64-port crossbars, and leveraging these as high-radix routers within NoCs. There are several other stand-alone NoC prototypes that also explored practical implementations with timing, power and area consideration, such as the 1 GHz Broadcast NoC [68] that optimizes for energy, latency and throughput using virtual bypassing and low-swing signaling for unicast, multicast, and broadcast traffic. Virtual bypassing is leveraged in the SCORPIO NoC.

To reduce the impact of snoop requests on the required network and tag lookup bandwidth, snoop filters remove impertinent requests. Two broad classes of snoop filters are source and destination filters. Destination filters reduce the snoop-induced tag lookups but do not address the broadcasts in the network. Thus, snoops are still sent to all nodes and filtered at the destinations prior to being sent to the processor core. The benefits are reduced tag lookup energy and bandwidth. Source filters reduce both the tag lookups and broadcast overhead. Broadcasts may not be necessary especially when there are only a few on-chip sharers of a cache block, and can be successfully filtered with source-based filters. In addition to these two categories of snoop filters, in-network coherence

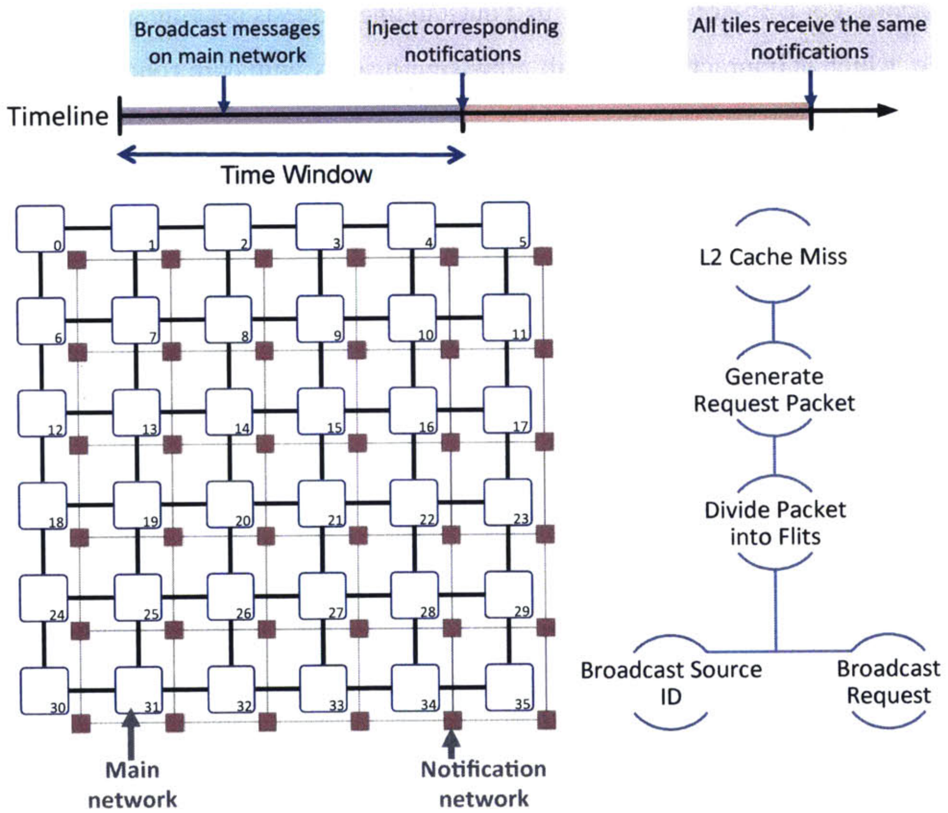


Figure 3-3: SCORPIO NoC's Two Physical Networks and Synchronized Time Window Injection Policy

filtering addresses this by embedding snoop filtering within the network. As requests traverse the network, they are effectively filtered such that only the cores caching that block will receive the snoop request. [11] The in-network filtering principles in [11] can be integrated with different ordered networks, but the integration may require surpassing additional challenges.

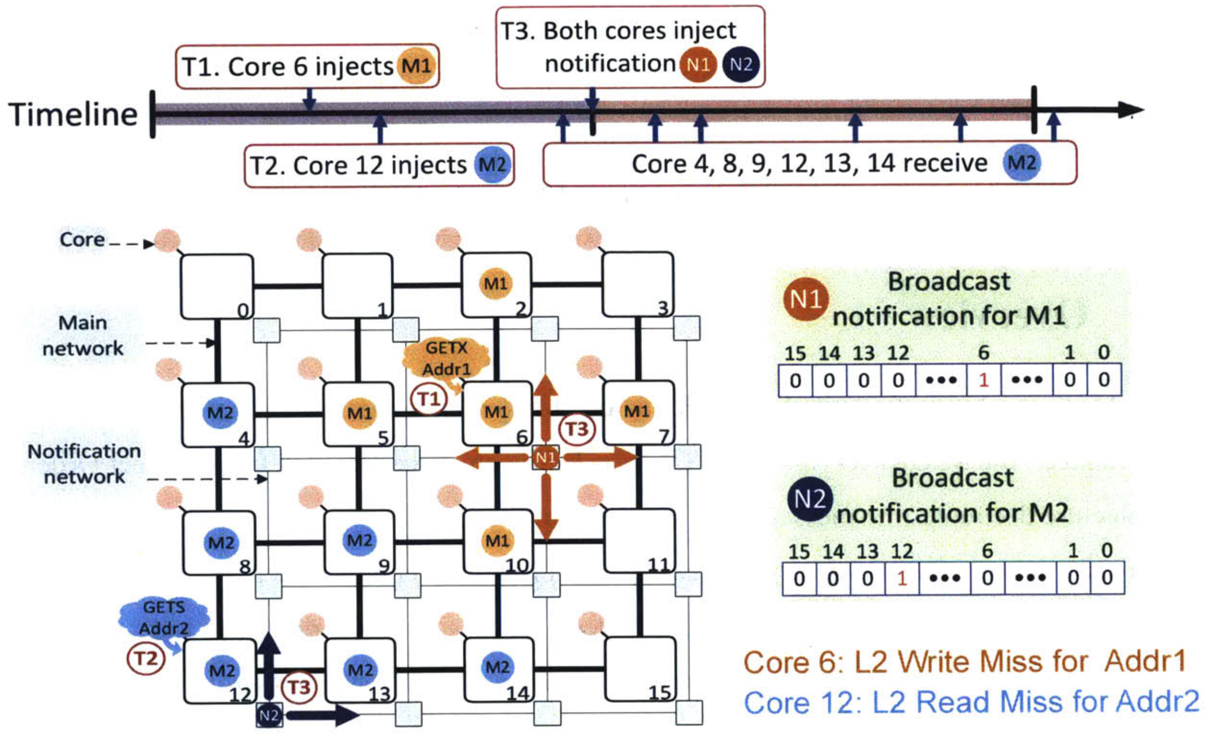
3.3 Overview

To tackle the problem of a centralized ordering point for ordering messages on an unordered on chip interconnect, the SCORPIO (Snoopy COherent Research Processor with Interconnect Ordering) architecture decouples the message ordering from message delivery using two physical networks, *main network* and *notification network*. Distributed ordering is enforced such that each node orders messages locally while maintaining a global order with consistent ordering rules and synchronized time windows. Each L2 cache miss generates a request packet which is further divided into flits for network injection, as seen in Figure 3-3. Coherent request messages are single-flit packets where the head flit alone suffices and holds all the information. Each coherent request is broadcast to all nodes on the main network. Through the notification network, all nodes are notified of broadcasts requests that were sent on the main network. Snoopy coherence protocols depend on the logical order which does not necessarily coincide with physical time ordering. Therefore, the global ordering mechanisms can be simplified to avoid maintaining expensive timestamps.

3.3.1 Decouple Message Ordering from Message Delivery

Each node in the system consists of a main network router, a notification router, as well as a network interface controller or logic interfacing the core/cache and the two routers. The network interface controller (NIC) encapsulates the coherence requests/responses from the core/cache and injects them into the appropriate virtual networks in the main network.

The main network is an unordered network and is responsible for broadcasting actual coherence requests to all other nodes and delivering the responses to the requesting nodes. Since the network is unordered, the broadcast coherence requests from different source nodes may arrive at the NIC of each node in any order. The NICs of the main network are then responsible for forwarding requests in global order to the cache controller, assisted by the notification network.



T1. Core 6 injects M1
T3. Both cores inject notification N1 N2
Timeline
T2. Core 12 injects M2
Core 4, 8, 9, 12, 13, 14 receive M2
Core
Main network
Notification network
GETX Addr1
GETS Addr2
N1 Broadcast notification for M1
15 14 13 12 6 1 0
0 0 0 0 ••• 1 ••• 0 0
N2 Broadcast notification for M2
15 14 13 12 6 1 0
0 0 0 1 ••• 0 ••• 0 0
Core 6: L2 Write Miss for Addr1
Core 12: L2 Read Miss for Addr2

Figure 3-4: SCORPIO 16-Node Walkthrough - Request and Notification Injection

For every coherence request sent on the main network, a notification message encoding the source node's ID (SID) is broadcast on the notification network to notify all nodes that a coherence request from this source node is in-flight and needs to be ordered. The notification network microarchitecture will be detailed later in Section 3.4.2; Essentially, it is a bit vector where each bit corresponds to a request from a source node, so broadcasts can be merged by OR-ing the bit vectors in a contention-less manner. The notification network thus has a fixed maximum network latency bound. Accordingly, we maintain synchronized time windows, greater than the latency bound, at each node in the system. We synchronize and send notification messages only at the beginning of each time window, thus guaranteeing that all nodes received the same set of notification messages at the end of that time window. By processing the received notification messages in accordance with a *consistent* ordering rule, all network interface controllers determine *locally* the *global* order for the actual coherence requests in the main network. As a result, even though the coherence requests can arrive at each NIC in any order, they are serviced at all nodes in the same order.

On the receive end, the NIC forwards the received coherence requests to the core/cache in accordance with the global order, which is determined using the received notification messages at the end of each time window. The NIC uses an *Expected Source ID* (ESID) register to keep track of and informs the main network router which coherence request it is waiting for. For example, if the ESID stores a value of 3, it means that the NIC is waiting for a coherence request from node 3 and would not forward coherence requests from other nodes to the core/cache. Upon receiving the request from node 3, the NIC updates the ESID and waits for the next request based on the global order determined using the received notification messages. The NIC forwards coherence responses to the core/cache in any order.

3.3.2 Walkthrough Example

A walkthrough example is useful for demonstrating how two messages are ordered globally with respect to each other. Core 6 has a L2 write miss for Address 1, and Core 12 has a L2 read miss for Address 2. Figure 3-4 shows at times T1 and T2, the cache controllers inject cache miss messages M1, M2 to the NIC at cores 6, 12 respectively. The NICs encapsulate these coherence requests into single flit packets, tag them with the SID of their source (6, 12 respectively), and broadcast them to all nodes in the main network. The notification is a one-hot bit vector with the bit asserted representing the SID of the injected request. At time T3, the start of the time window, notification messages N1 and N2 are generated corresponding to M1 and M2, and sent into the notification network.

Figure 3-5 shows the notification messages broadcast at the start of a time window are guaranteed to be delivered to all nodes by the end of the time window (T4). At this stage, all nodes process the notification messages received and perform a local but consistent decision to order these messages. In SCORPIO, we use a rotating priority arbiter to order messages according to increasing SID – the priority is updated each time window ensuring fairness. In this example, all nodes decide to process M2 before M1.

The decided global order is captured in the ESID register in NIC. In this example, ESID is currently 12 – the NICs are waiting for the message from core 12 (*i.e.* M2).

At time T5, when a coherence request arrives at a NIC, the NIC performs a check of its source ID (SID). If the SID matches the ESID then the coherence request is processed (*i.e.* dequeued, parsed and handed to the cache controller) else it is held in the NIC buffers. Once the coherence

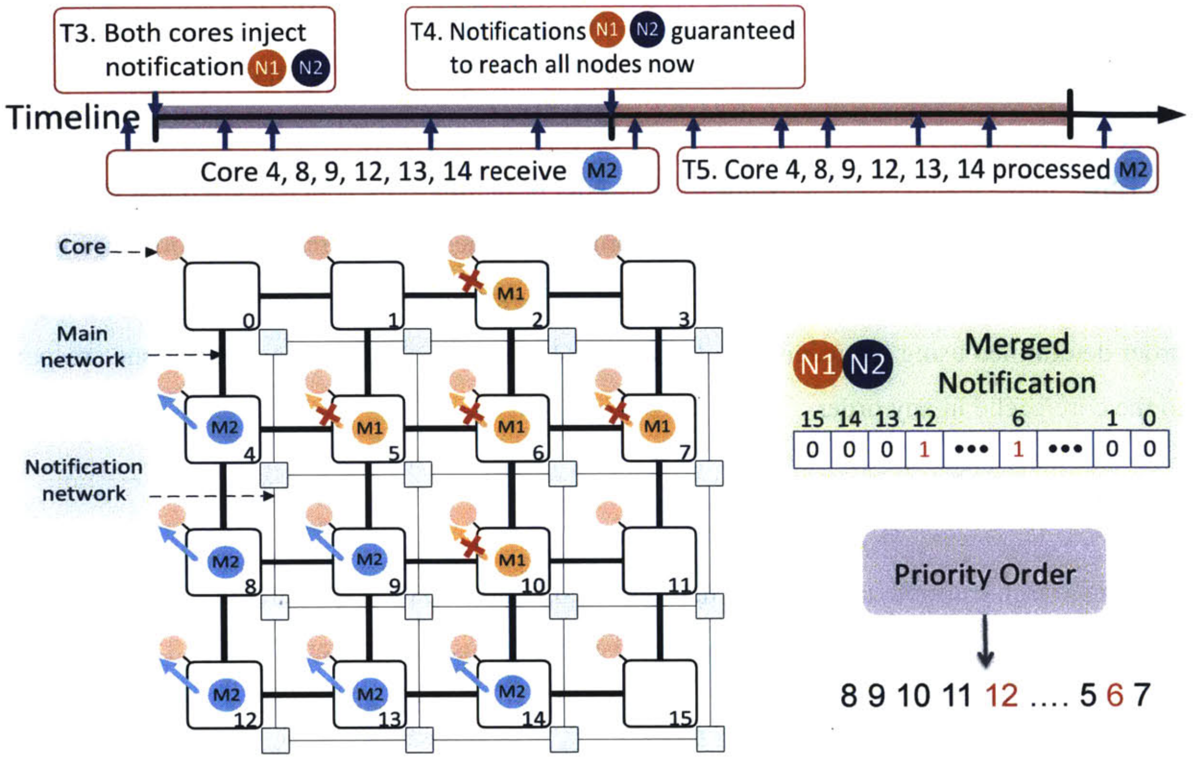

T3. Both cores inject notification N1 N2
T4. Notifications N1 N2 guaranteed to reach all nodes now
Timeline
Core 4, 8, 9, 12, 13, 14 receive M2
T5. Core 4, 8, 9, 12, 13, 14 processed M2
Core
Main network
Notification network
M1
M2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
N1 N2 Merged Notification
15 14 13 12 6 1 0
0 0 0 1 ••• 1 ••• 0 0
Priority Order
8 9 10 11 12 .... 5 6 7

Figure 3-5: SCORPIO 16-Node Walkthrough - Message Ordering

request with the SID equal to ESID is processed, the ESID is updated to the next value (based on the notification messages received). In this example, the NIC has to forward M2 before M1 to the cache controller. If M1 arrives first, it will be buffered in the NIC (or router, depending on the buffer availability at NIC) and wait for M2 to arrive.

In Figure 3-6, Cores 9 and 0 respond to M1 (at T7) and M2 (at T6) respectively. The data responses are sent on the unordered response virtual network within the main network. Thus data responses are unicast messages and do not require ordering. All cores thus process all request messages in the same order, *i.e.* M2 followed by M1.

3.4 Microarchitecture

To achieve distributed global ordering, at low cost, we utilize two physically separate mesh networks: main and notification networks. Routers of both networks connect to a single network interface at each node. To ensure ordering of requests, the notification network is a fixed-latency, contention-

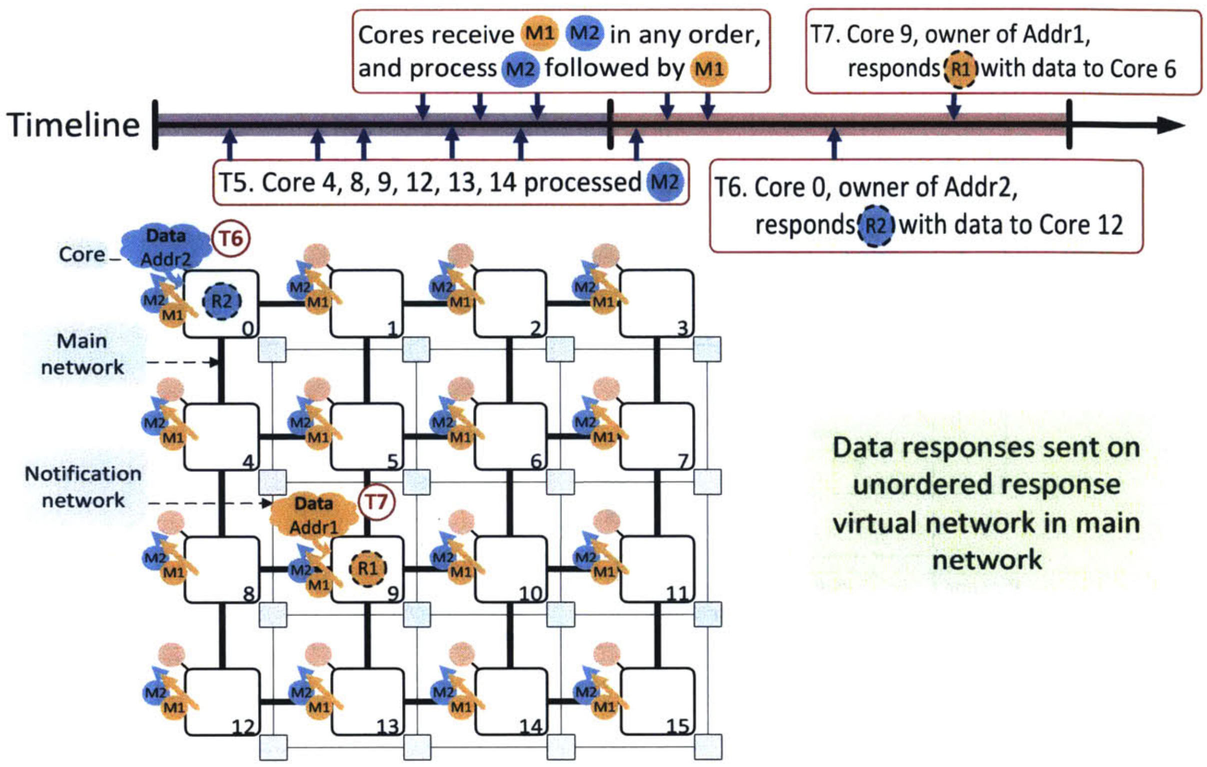

Cores receive M1 M2 in any order, and process M2 followed by M1
T7. Core 9, owner of Addr1, responds R1 with data to Core 6
Timeline
T5. Core 4, 8, 9, 12, 13, 14 processed M2
T6. Core 0, owner of Addr2, responds R2 with data to Core 12
Core
Data Addr2
T6
Main network
Notification network
Data Addr1
T7
Data responses sent on unordered response virtual network in main network

Figure 3-6: SCORPIO 16-Node Walkthrough - Data Responses

free, bufferless network, that informs all nodes of broadcast requests sent on the main network. The main network accepts requests and delivers them to the destinations in any order and at any time. For deadlock-avoidance and performance reasons, an express path is set up for the highest priority request.

3.4.1 Main Network

Figure 3-7 shows the microarchitecture of the three-stage main network router. During the first pipeline stage, the incoming flit is buffered (BW), and in parallel arbitrates with the other virtual channels (VCs) at that input port for access to the crossbar's input port (SA-I). In the second stage, the winners of SA-I from each input port arbitrate for the crossbar's output ports (SA-O), and in parallel obtain a free VC at the next router if possible (VA). In the final stage, the winners of SA-O traverse the crossbar (ST). Next, the flits traverse the link to the adjacent router in the following cycle.

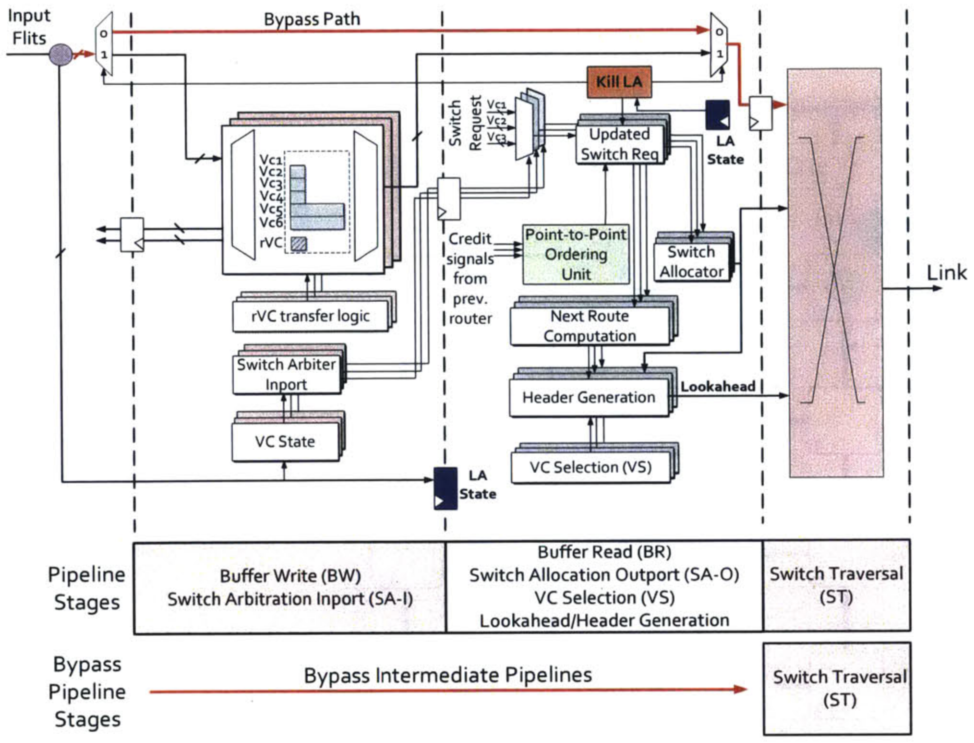


Figure 3-7: Router Microarchitecture

Single-Cycle Pipeline Optimization

To reduce the network latency and buffer read/write power, we implement lookahead (LA) bypassing [53, 68]; a lookahead containing control information for a flit is sent to the next router during that flit's ST stage. At the next router, the lookahead performs route-computation and tries to preallocate the crossbar for the approaching flit. Lookaheads are prioritized over buffered flits[1] – they attempt to win SA-I and SA-O, obtain a free VC at the next router, and setup the crossbar for the approaching flits, which then bypass the first two stages and move to ST stage directly. Conflicts between lookaheads from different input ports are resolved using a static, rotating priority scheme. If a lookahead is unable to setup the crossbar, or obtain a free VC at the next router, the incoming flit is buffered and goes through all three stages. The control information carried by lookaheads is already included in the header field of conventional NoCs – destination coordinates, VC ID and the output port ID – and hence does not impose any wiring overhead.

[1]Only buffered flits in the reserved VCs, used for deadlock avoidance, are an exception, prioritized over lookaheads.

Single-Cycle Broadcast Optimization

To alleviate the overhead imposed by the coherence broadcast requests, routers are equipped with single-cycle multicast support [68]. Instead of sending the same requests for each node one by one into the main network, we allow requests to fork through multiple router output ports in the same cycle. Broadcast flits are routed through the network along a dimension-ordered XY-tree, where they are potentially granted multiple output ports during switch allocation at each router. It dramatically reduces network contention and provides efficient hardware broadcast support.

Deadlock Avoidance

The network enforces sequential consistency for coherent requests but also supports message classes to handle data responses. The main network contains multiple virtual networks to avoid protocol-level deadlocks while servicing different message types. The three virtual networks are as follows.

- **Globally Ordered Request (GO-REQ):** Delivers coherence requests, and provides global ordering, lookahead-bypassing and hardware broadcast support. The NIC processes the received requests from this virtual network based on the order determined by the notification network.

- **Point-to-point Ordered Request (P2P-REQ):** Delivers non-coherent requests to the memory controllers. The memory controller processes the received responses in the order of arrival, as the network guarantees non-coherent memory accesses from the same processor are executed in program order, with point-to-point ordering support.

- **Unordered Response (UO-RESP):** Delivers coherence responses, and supports lookahead-bypassing for unicasts. The NIC processes the received responses in any order.

The main network uses XY-routing algorithm which ensures deadlock-freedom for the UO-RESP virtual network. For the GO-REQ virtual network, however, the NIC processes the received requests in the order determined by the notification network which may lead to deadlock; the request that the NIC is awaiting might not be able to enter the NIC because the buffers in the NIC and

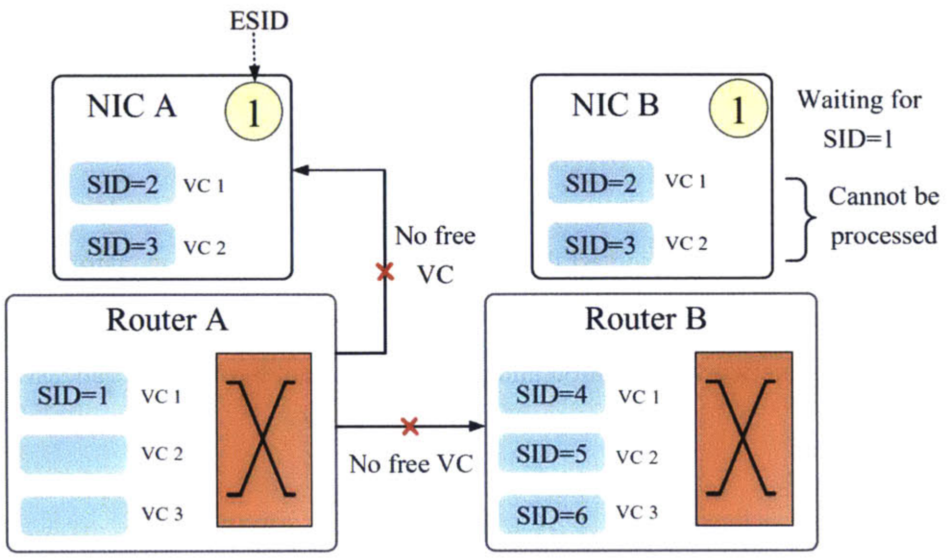


Figure 3-8: Deadlock Scenario in GO-REQ Virtual Network

routers en route are all occupied by other requests. Figure 3-8 shows a scenario such that a flit with $SID = 1$ is unable to progress due to unavailable VC space in the neighboring router and local NIC.

To prevent the deadlock scenario, we add one reserved virtual channel (rVC) to each router and NIC, reserved for the coherence request with SID equal to ESID that the NIC, at that router, is waiting for. Figure 3-9 shows that the reserved VC creates a deadlock-free, "escape", path for the highest priority flit to reach the destinations. Thus, we can ensure that the requests can always proceed toward the destinations.

Consider a mesh NoC architecture in which, (i) M VCs per input port for the GO-REQ virtual network, (ii) 1 reserved virtual channel in present in the NICs and in each input port of the router. Deadlock is avoided when the reserved VC can only be occupied by the highest priority flit - flit earliest in global order.

Proof. Suppose there is a deadlock in the network and the highest priority flit, flit earliest in global order, is unable to make progress. Let flit F be the highest priority flit, at router R with $ESID = E$. If the flit is unable to make progress it implies either *(a)* F is unable to go up to the NIC at router R, or *(b)* F is unable to proceed to a neighboring router S.

Since F is the highest priority flit, it must have SID equal to ESID of the router R because a

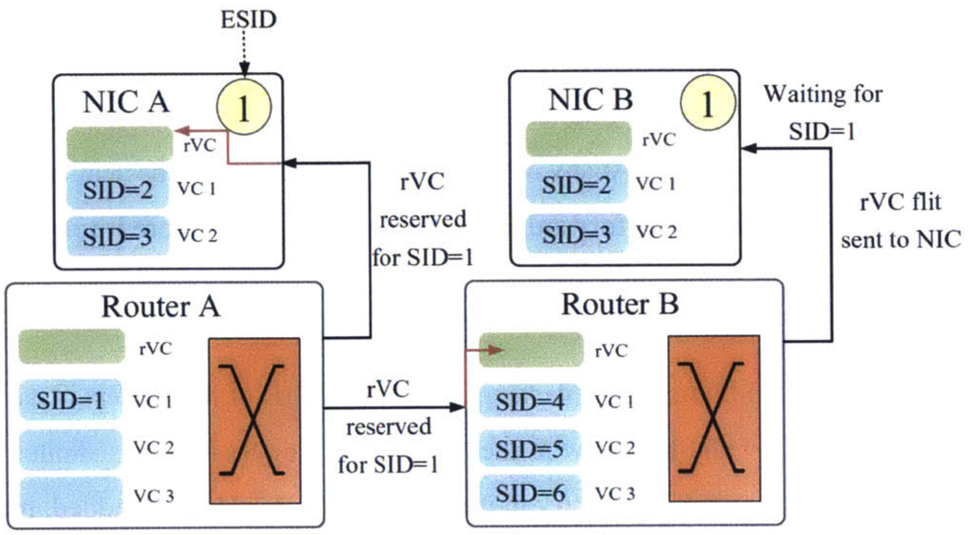


Figure 3-9: Deadlock Avoidance in GO-REQ Virtual Network with Reserved VC

lower priority ESID is only obtained if the higher priority flit has been received at the NIC. Since a rVC is available for F in the NIC, flit F can be sent to the NIC attached to router R.

Flit F can not proceed to router S if the rVC and other VCs are full. The rVC is full if router S has an ESID with a higher priority than E. This is not possible because F is the highest priority flit which implies any flit of higher priority has already been received at all nodes in the system. For $E1$ with lower or same priority as E, the rVC is available and flit F can make progress. Thus, there is a contradiction and we can ensure that the requests can always proceed toward the destinations. □

Point-to-Point Ordering for GO-REQ

In addition to enforcing a global order, requests from the *same* source also need to be ordered with respect to each other. Since requests are identified by source ID alone, the main network must ensure that a later request does not overtake an earlier request from the same source. To enforce this in SCORPIO, the following property must hold: *Two requests at a particular input port of a router, or at the NIC input queue cannot have the same SID.* Coupled with deterministic routing, this property guarantees point-to-point ordering of all flits between any source-destination pair in the network.

Consider a particular source-destination pair $A - B$. All flits sent from A to B follow the same path, say, $\Pi = \{A, r_0, r_1, \cdots, r_n, B\}$, where r_i's represent the routers on the path. Let flit i be inserted by A at time t_i ($i \in \mathbb{Z}$ and $t_i < t_j \forall i < j$).

Suppose $i < j$. Flit j may enter the local port of the router attached to the NIC of source A, only after flit i has left the local port. Similarly flit j may be sent to destination B by router r_n, only after flit i has been processed at B. At any intermediate router, flit j may be sent from router r_k to router r_{k+1} only after flit i has been forwarded from r_{k+1} to r_{k+2}. Therefore it follows that flit i is processed at destination B before flit j, for any $i < j$, i.e. $t_i < t_j$. Hence point-to-point ordering is maintained.

At each output port, a SID tracker table keeps track of the SID of the request in each VC at the next router. Suppose a flit with SID = 5 wins the north port during SA-O and is allotted VC 1 at the next router in the north direction. An entry in the table for the north port is added, mapping (VC 1) $\rightarrow$ (SID = 5). At the next router, when flit with SID = 5 wins all its required output ports and leaves the router, a credit signal is sent back to this router and then the entry is cleared in the SID tracker. Prior to the clearance of the SID tracker entry, any request with SID = 5 is prevented from placing a switch allocation request.

3.4.2 Notification Network

The notification network is an ultra-lightweight bufferless mesh network consisting of 5 N-bit bitwise-OR gates and 5 N-bit latches at each "router" as well as N-bit links connecting these "routers", as shown in Figure 3-10, where N is the number of cores. A notification message is encoded as a N-bit vector where each bit indicates whether a core has sent a coherence request that needs to be ordered. With this encoding, the notification router can merge two notification messages via a bitwise-OR of two messages then forward the merged message to the next router.

Time Window Network Injection Policy

At the beginning of a time window, a core that wants to send a notification message asserts its associated bit in the bit-vector and sends the bit-vector to its notification router. Every cycle, each notification router merges received notification messages and forwards the updated message to all its neighbor routers in the same cycle. Since messages are merged upon contention, messages can always proceed through the network without being stopped, and hence, no buffer is required and network latency is bounded. At the end of that time window, it is guaranteed that all nodes in the network receive the same merged message, and this message is then sent to the NIC for further processing to determine the global order of the corresponding coherence requests in the main network. For example, if node 0 and node 6 want to send notification messages, at the beginning of a time window, they send the messages with bit 0 and bit 6 asserted, respectively, to their notification routers. At the end of the time window, all nodes receive a final message with both bits 0 and 6 asserted. In a 6×6 mesh notification network, the maximum latency is 6 cycles along the X dimension and another 6 cycles along Y, so the time window is set to 13 cycles.

Multiple Requests per Notification Message

Thus far, the notification message described handles one coherence request per node every time window, *i.e.* only one coherence request from each core can be ordered within a time window. However, this is inefficient for more aggressive cores that have more outstanding misses, especially for bursty traffic. With a time window of $2k$ in a k×k mesh network, and M single-flit packets in the burst, the last flit must wait at least $2Mk$ cycles until it is sent into the network. For example, when an aggressive core generates 6 requests at around the same time, the last request can only be ordered at the end of the 6th time window, incurring latency overhead. To resolve this, instead of

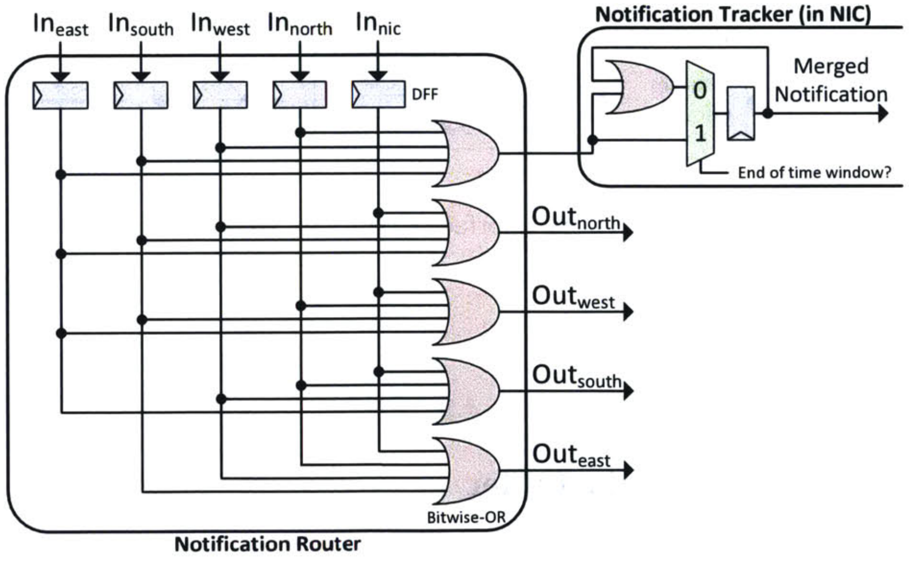


Figure 3-10: Notification Router Microarchitecture

using only 1 bit per core, we dedicate multiple bits per core to encode the number of coherence requests that a core wants to order in this time window, at a cost of larger notification message size. For example, if we allocate two bits instead of 1 per core in the notification message, the maximum number of coherence requests that can be ordered in this time window can be increased to 3. The number of coherence requests is encoded in binary, where a value of 0 means no request to be ordered, 1 implies 1 request, while 3 indicates 3 requests to be ordered (maximum value that a 2-bit number can represent). Now, the core sets the associated bits to the number of coherence requests to be ordered and leaves other bits as zero. This allows us to continue using the bitwise-OR to merge the notification messages from other nodes.

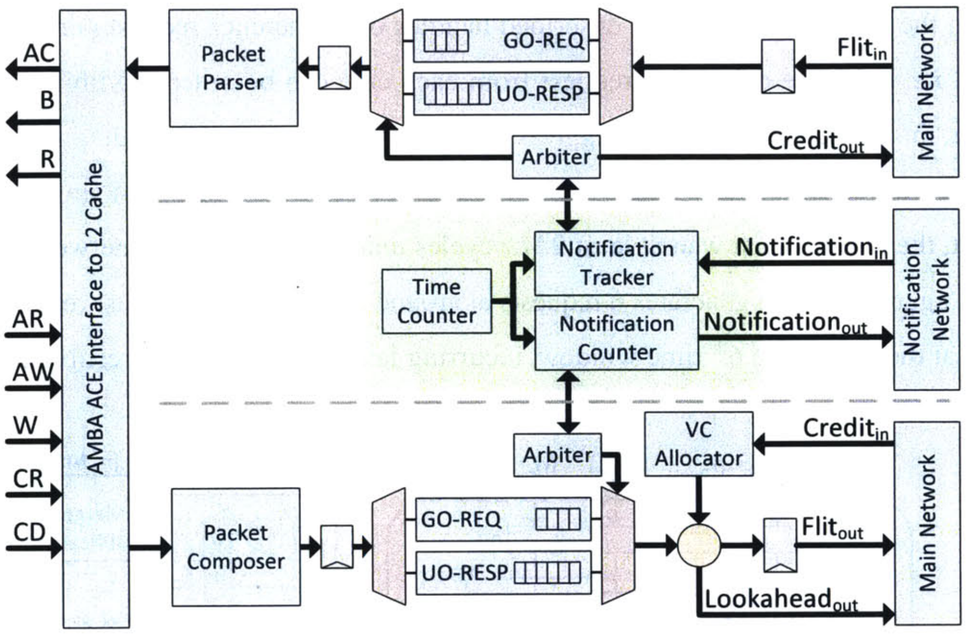


Figure 3-11: Network Interface Controller Microarchitecture

3.4.3 Network Interface Controller

Figure 3-11 shows the microarchitecture of the NIC, which interfaces between the core/cache and the main and notification network routers.

Sending Notifications

On receiving a message from core/cache, the NIC encapsulates the message into a packet and sends it to the appropriate virtual network. If the message is a coherence request, the NIC needs to send a notification message so that the coherence request can be ordered. Since the purpose of the notification network is to decouple the coherence request ordering from the request delivery, the NIC can always send the coherence requests to the main network whenever possible and send the corresponding notification messages at the beginning of later time windows. We use a counter to keep track of how many pending notification messages still remain to be sent. The counter can be sized arbitrarily for expected bursts; when the maximum number of pending notification messages, represented by this counter, is reached, the NIC blocks new coherence requests from injecting into the main network.

Receiving notifications.

At the end of every time window, the NIC pushes the received merged notification message into the notification tracker queue. When the notification tracker queue is not empty and there is no previously read notification message being processed, the head of the queue is read and passed through a rotating priority arbiter to determine the order of processing the incoming coherence requests (*i.e.* to determine ESIDs). On receiving the expected coherence request, the NIC parses the packet and passes appropriate information to the core/cache, and informs the notification tracker to update the ESID value. Once all the requests indicated by this notification message are processed, the notification tracker reads the next notification message in the queue if available and re-iterate the same process mentioned above. The rotating priority arbiter is updated at this time.

If the notification tracker queue is full, the NIC informs other NICs and suppresses other NICs from sending notification messages. To achieve this, we add a "stop" bit to the notification message. When any NIC's queue is full, that NIC sends a notification message with the "stop" bit asserted, which is also OR-ed during message merging; consequently all nodes ignore the merged notification message received; also, the nodes that sent a notification message this time window will resend it later. When this NIC's queue becomes non-full, the NIC sends the notification message with the "stop" bit de-asserted. All NICs are enabled again to (re-)send pending notification messages when the "stop" bit of the received merged notification message is de-asserted.

3.5 Architecture Analysis

For performance comparison with prior in-network coherence proposals, we use the GEMS Simulator [59] with the GARNET network model. GARNET [10] captures the detailed, cycle-accurate behavior of the on-chip network, yielding accurate simulation results. A 36-core system is simulated, where each tile consists of an in-order SPARC processor, 32 kB I&D caches, and 128 kB private L2 cache. The on-chip network is a 6×6 mesh, modeled in detail to contain all the mechanisms of the SCORPIO network. We vary the on-chip network design parameters, i.e channel-width, VC count, simultaneous notifications and view the impact on the performance.

The on-chip network RTL is fully verified with synthetic traffic patterns: (1) uniform random (random source and destination selection with equal probability for all nodes), (2) local neighbor (send packets locally to neighboring nodes), and (3) broadcast traffic (1 to all multicast traffic where nodes are randomly selected to broadcast). All packets are received at the destinations and the GO-REQ virtual network properly orders requests globally. The most accurate latency-throughput characteristics are obtained from the RTL of the network. Uniform random unicast and broadcast traffic is injected for UO-RESP/P2P, and GO-REQ, virtual networks respectively.

The SCORPIO router and network interface are synthesized with Synopsis Design Compiler for the IBM 45nm technology node. We obtain the area and power overheads from the post-synthesis generated logs.

3.5.1 NoC Parameter Sweep

In GEMS, we sweep several key SCORPIO network parameters, channel-width, number of VCs, and number of simultaneous notifications, to arrive at the final 36-core fabricated configuration. Channel-width impacts network throughput by directly influencing the number of flits in a multi-flit packet, affecting serialization and essentially packet latency. The number of VCs also affects the throughput of the network and application runtimes, while the number of simultaneous notifications affect ordering delay. Figures 3-12 and 3-13 show the variation in runtime as the channel-width and number of VCs are varied. All results are normalized against a baseline configuration of 16-byte channel-width and 4 VCs in each virtual network.

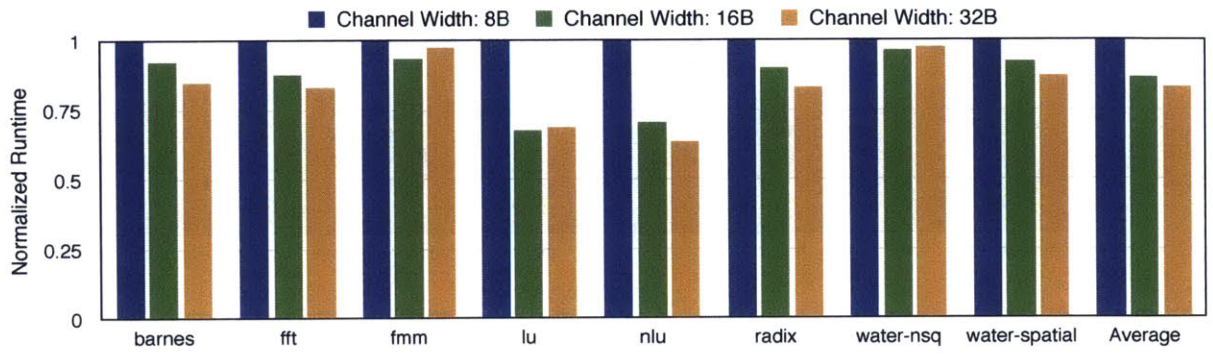


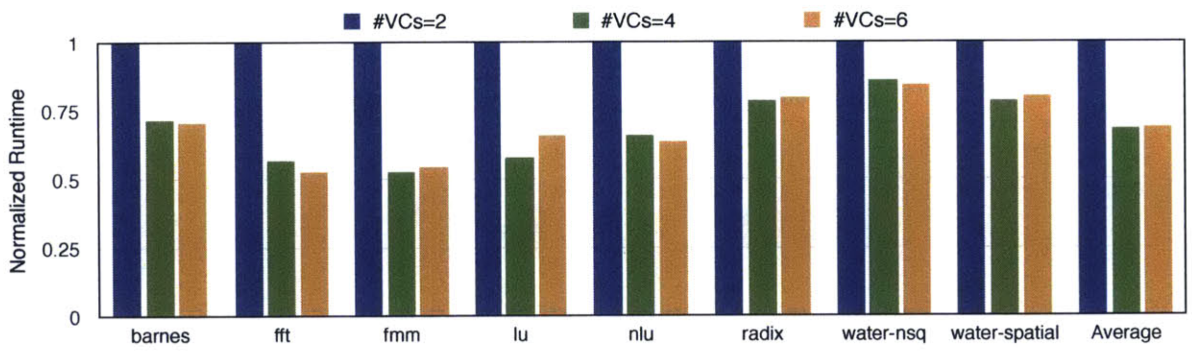
Figure 3-12: Impact of Varying the NoC Channel-Width

Channel-Width

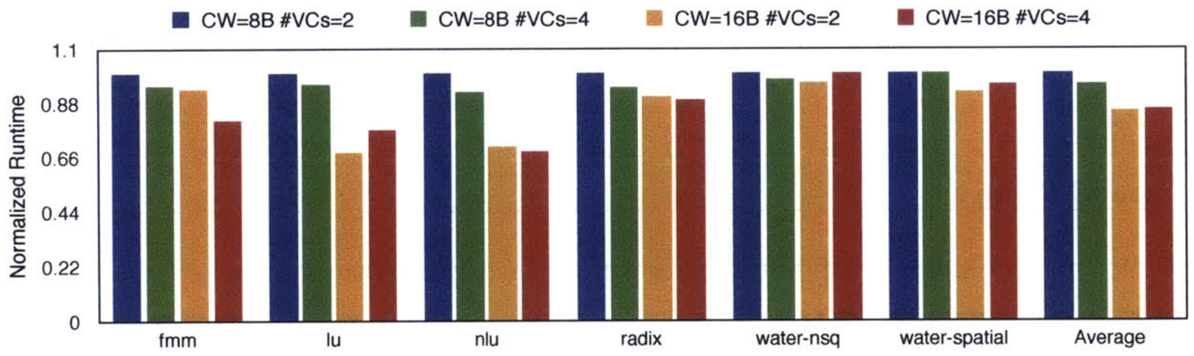
While a larger channel-width offers better performance, it also incurs greater overheads – larger buffers, higher link power and larger router area. A channel-width of 16 bytes translates to 3 flits per packet for cache line responses on the UO-RESP virtual network. A channel-width of 8 bytes would require 5 flits per packet for cache line responses, which degrades the runtime for a few applications. While a 32 byte channel offers a marginal improvement in performance, it expands router and NIC area by 46%. In addition, it leads to low link utilization for the shorter network requests. The 36-core chip contains 16-byte channels due to area constraints and diminishing returns for larger channel-widths.

Number of Virtual Channels

Two VCS provide insufficient bandwidth for the GO-REQ virtual network which carries the heavy request broadcast traffic. Besides, one VC is reserved for deadlock avoidance, so low VC configurations would degrade runtime severely. There is a negligible difference in runtime between 4 VCs and 6 VCs. Post-synthesis timing analysis of the router shows negligible impact on the operating frequency as the number of VCs is varied, with the critical path timing hovering around 950ps. The number of VCs indeed affects the SA-I stage, but it is off the critical path. The VC selection for the downstream routers does not affect the critical path as well because the VS mechanism simply consists of choosing from a pool of free VCs that are already latched. However,



(a) GO-REQ VCs



(b) UO-RESP VCs

Figure 3-13: Virtual Channel Count Sweep for GO-REQ and UO-RESP Virtual Networks

a tradeoff of area, power, and performance still exists. Post-synthesis evaluations show 4 VCs is 15% more area efficient, and consumes 12% less power than 6 VCs. Hence, our 36-core chip contains 4 VCs in the GO-REQ virtual network. For the UO-RESP virtual network, the number of VCs does not seem to impact run time greatly once channel-width is fixed. UO-RESP packets are unicast messages, and generally much fewer than the GO-REQ broadcast requests. Hence 2 VCs suffices. The P2P-REQ virtual network is also expected to have low load, carrying unicast messages to the memory controller, hence we went with 2 VCs.

Number of Simultaneous Notifications

The Freescale e200 cores used in our 36-core chip are constrained to two outstanding messages at a time because of the AHB interfaces at its data and instruction cache miss ports. Due to the low injection rates, we choose a 1-bit-per-core (36-bit) notification network which allows 1 notification per core per time window.

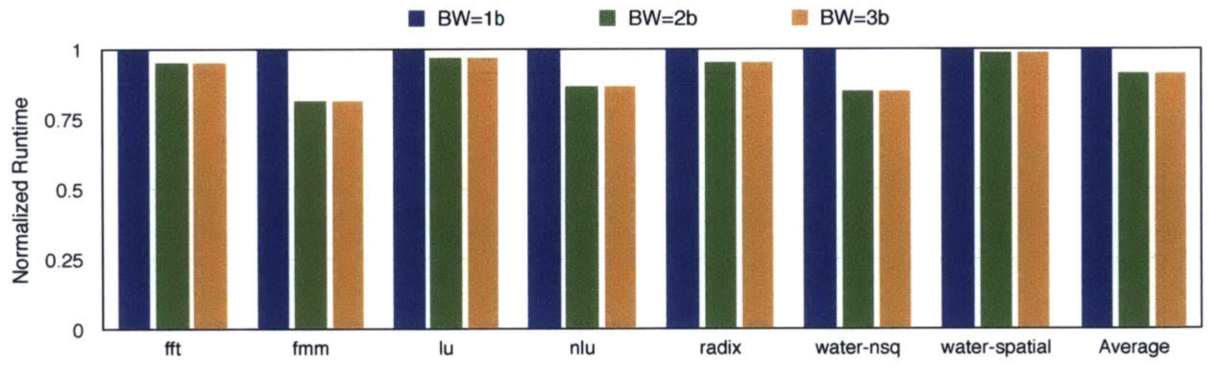


Figure 3-14: Impact of Varying the Number of Simultaneous Notifications

We evaluate if a wider notification network that supports more notifications each time window will offer better performance. Supporting 3 notifications per core per time window, will require 2 bits per core, which results in a 72-bit notification network. Figure 3-14 shows 36-core GEMS simulations of SCORPIO achieving 10% better performance for more than one outstanding message per core with a 2-bit-per-core notification network, indicating that bursts of 3 messages per core occur often enough to result in overall runtime reduction. However, more than 3 notifications per time window (3-bit-per-core notification network) does not reap further benefit, as larger bursts of messages are uncommon. A notification network data width scales as $O(m \times N)$, where m is the number of notifications per core per time window. Our 36-bit notification network has $< 1\%$ of tile area and power overheads; Wider data widths only incurs additional wiring which has minimal area/power compared to the main network and should not be challenging given the excess wiring space remaining in the chip.

3.5.2 Performance Comparison with Prior Proposals

To compare SCORPIO's performance with TokenB and INSO, we ran a subset of benchmarks on a 16-core system in GEMS with cycle-accurate network modeling. Figure 3-15 shows the normalized runtime when keeping all conditions equal besides the ordered network. The difference in performance between SCORPIO and TokenB is negligible. However, we do not model the behavior of TokenB in the event of data races where retries and expensive persistent requests are generated. These persistent requests create network traffic and contention, degrading overall performance. Thus, the best case performance of the baseline TokenB approach is depicted.

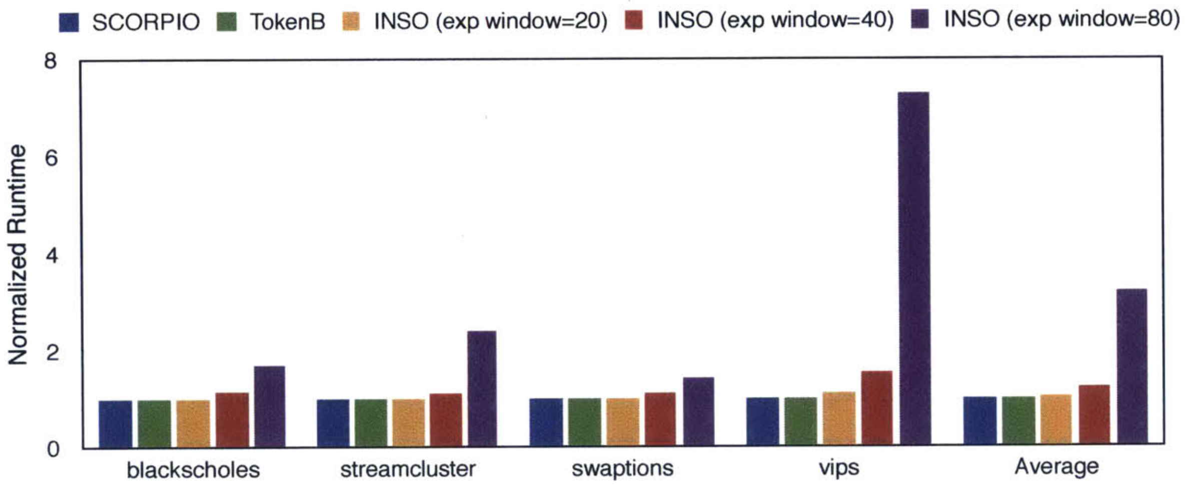


Figure 3-15: Comparison with TokenB and INSO

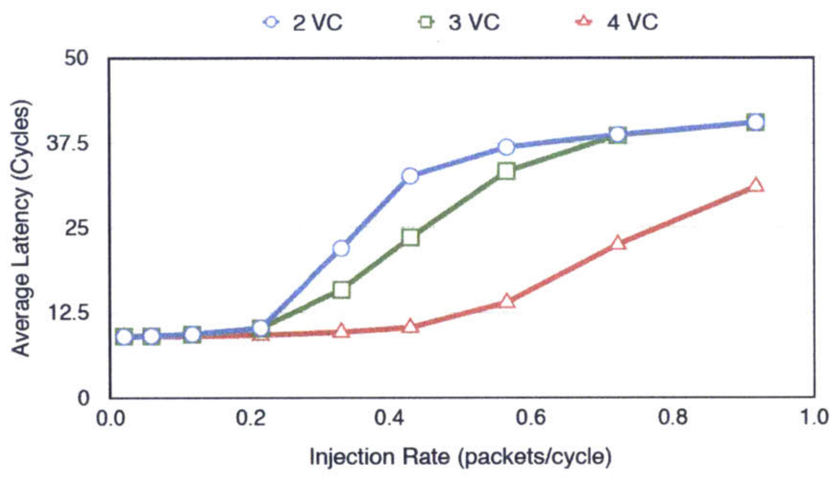
SCORPIO's runtime is 19.3% and 70% less than INSO with an expiration window of 40 and 80 cycles, respectively. Higher expiration window lengths influence longer wait times for flits at the destinations. Once an expiration for the prioritized source ID is received from the network, the NIC updates the prioritized source ID to the next one in order. This goes on until all the flits are processed in order. Shorter expiration windows lead to increased network traffic when flits are not drained efficiently and already new expiry messages are sent. SCORPIO performs as well as TokenB without persistent requests and INSO with an impractical expiration window size of 20 cycles.

3.5.3 Performance with Synthetic Traffic

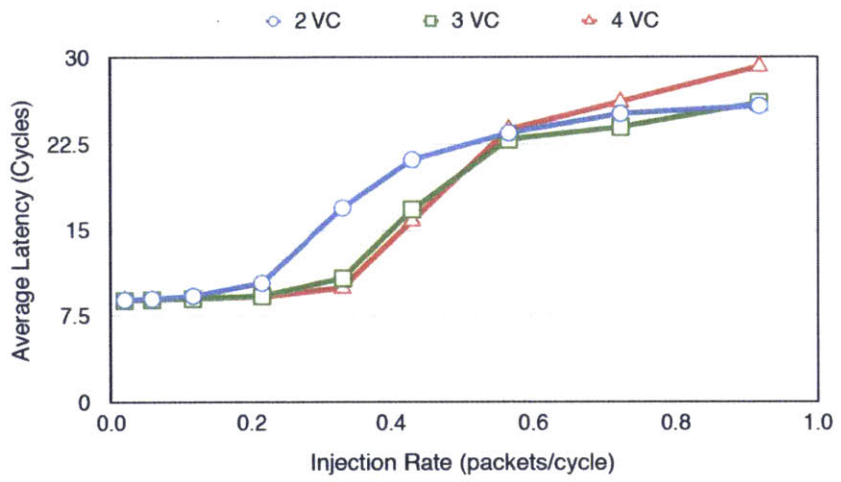
The latency and throughput is obtained from the SCORPIO RTL for a 36 node network. Since the main network consists of three virtual networks with different properties and message types, we dive into the performance of each.

UO-RESP Virtual Network Latency and Throughput

The UO-RESP network handles unicast traffic without ordering guarantees. Under uniform random traffic injection with single-flit packets, the latency-throughput curves for the UO-RESP network is shown, in Figure 3-16a as the number of VCs are varied. For low loads, all curves exhibit a zero load latency of approximately 10 cycles. Increasing the number of VCs benefits the throughput, which is necessary at higher injection rates. With 2 and 3 VCs, the network saturates quickly and levels off due to finite destination buffering in the NICs. Full system application suites, SPLASH-2 and PARSEC, generates network traffic that usually falls within the low injection rate range. Design exploration, discussed in Section 3.5.1, with these applications revealed that 2 VCs suffices for the UO-RESP network.



(a) UO-RESP Virtual Network



(b) P2P Virtual Network

Figure 3-16: Network Performance for Point-to-Point Requests and Unordered Responses

P2P-REQ Virtual Network Latency and Throughput

Uniform random traffic, in the form of single-flit P2P packets, are generated and sent into the P2P virtual network. The plot in Figure 3-16b shows the latency as the number of VCs are varied. The latency is 8 cycles at low loads, which saturates quickly despite the number of VCs. The point-to-point ordering mechanism in SCORPIO prevents multiple flits from the same source, from residing in the same router, at the same time. A flit waits in the router until there isn't a chance of conflict at the downstream router, with another flit from the same source. Due to the low injection rates of applications, and limited non-coherent communication, the P2P virtual networks does not need more than 2 VCs.

GO-REQ Virtual Network Latency and Throughput

The GO-REQ virtual network handles broadcast requests and quickly saturates with increasing injection rate. The ordering at the destinations and finite destination buffers, cause buffer back-pressure and the network gets congested quickly leading to higher network latencies. The worst case condition is when every node sends a flit into the network, this time-window. The destinations can thus only service one flit per source, every N cycles, where N is the number of nodes. As the time-window is less than N cycles, the injection of flits the following time-window exacerbates the congestion, and degrades performance. The notification "stop" signal alleviates this, as the destinations indicate whether new traffic should be allowed into the network. We evaluate the performance of the network with broadcast, unicast and point-to-point messages, where all are single-flit packets, and the percentage of broadcasts in the network changes while the injection rate is kept constant. Figure 3-17 shows the latency curve as the percentage of network traffic that are broadcasts is varied, while maintaining a overall fixed injection rate for all request types.

The average hop count is $\sqrt{k}/2$ hops on a k$\times$k mesh, which results in an average 3 cycle network latency, when the single-cycle lookahead bypassing path is taken per hop. An additional two cycles are incurred for network injection and ejection. Thus, on average the network latency at low load on the GO-REQ network is 5 cycles, but the ordering delay at the destinations affect the average latency. On average, a coherent request's latency is the sum of the time for notification to reach all destinations and be processed, and the average ordering delay. The notifications arrive

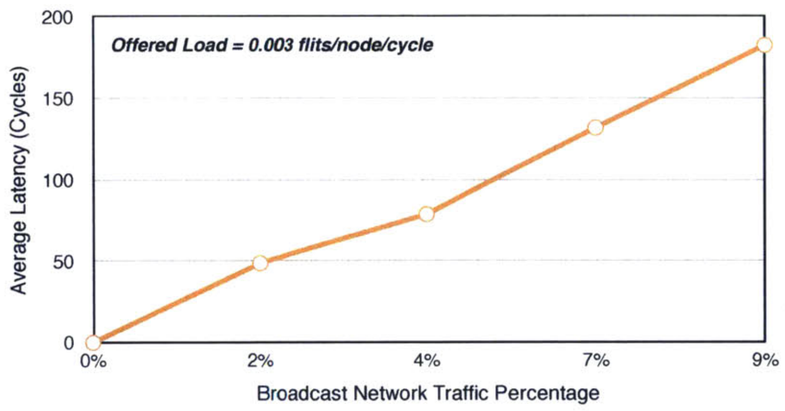


Figure 3-17: Average Latency for Broadcast Messages as its Fraction of Network Traffic is Varied

within one time-window and the order is determined then. However, the flit could have been sent earlier than the notification, but within the time-window. Thus, the maximum additional latency is 13 cycles for the 6×6 mesh, with an average latency estimated at 6 cycles. For low loads, the GO-REQ network has a 30 cycle latency which includes the ordering delay. With an average base latency of 19 cycles, the ordering delay is approximately 11 cycles at low load.

Table 3.1: Request Categories

Category	Data Location	Sufficient Permission	Trigger Condition
Local	Requester cache	Yes	Load hit and store hit (in M-state)
Local Owner	Requester cache	No	Store hit (in O-state)
Remote	Other cache	No	Load miss and store miss
Memory	Memory	No	Load miss and store miss

3.5.4 RTL Simulation Results

An analysis of the barnes SPLASH-2 application on the RTL identifies the latency breakdown as the requests from an application traverse through the SCORPIO NoC. The effect of different request types on the average latency is quantified for insight on the application's communication characteristics and the corresponding network performance. A snoopy coherent L2 cache controller is attached to the network, such that a memory trace injector feeds the benchmark traces gathered from GEMS, into the controller. The traces are obtained from the parallel portion of each workload.

The memory is modeled to be fully-pipelined with a fixed latency. Trace-driven simulations run for 400 K cycles, with a cache warm-up window of 20 K cycles.

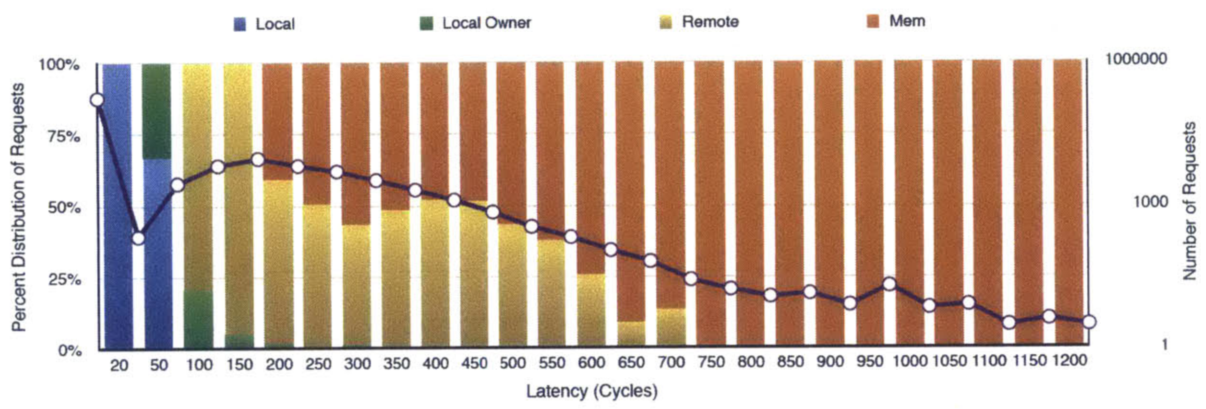


Figure 3-18: L2 Service Time (*Barnes*)

To dive into how different types of requests contribute to the average latency, we classify requests into 4 categories, shown in Table 3.1 *Local* requests hit in the cache and do not generate any network requests. *Local owner* requests occur on a store-hit in the Owned state, even though the local cache has valid data, it needs to send the request to the network and wait until the request is globally ordered before upgrading to the Modified state and completing the store. *Remote* requests are sent to the network as the data required resides in another cache on-chip. *Memory* requests are similar to *Remote* requests, but the data is in main memory and not cached on-chip.

Figure 3-18 shows the latency distribution of each request category for *barnes*. The *Memory* requests involve memory access latency and network latency, and forms the tail of the distribution. Because the L2 access latency is lower than the memory access latency, the overall latency for *Remote* requests is 200 cycles on average. Spatial locality in the memory traces lead to 81% hits in the requester cache, so even though latency is relatively high for *Remote* and *Memory* requests, average latency is around 51 cycles, still close to expected zero-load latency of 23 cycles.

The latency breakdown of each request category for the *barnes* benchmark traces is in Figure 3-19, For *Local* requests, as data resides in the local cache, only local L2 contributes to the round-trip latency, with an average latency of 12 cycles , which is the queuing latency and its zero-load latency. For *Local owner* requests, the delay in the router and NIC is due to the ordering delay of this request with respect to all other requests. For *Remote* requests, where the valid permission and data is in another cache, the latency involves (1) the request travel time through the network and ordering time at the remote cache (Local NIC+Network+Remote NIC), (2) the processing time to generate the response (Remote L2), and (3) response travel time through the network (Remote NIC+Network+Local NIC) and processing by the requester cache (Local L2). *Memory* requests are similar, except that valid permission and data resides in main memory, hence requests go through the interface between the memory controller and network (MIC NIC), and memory controller (MIC) instead.

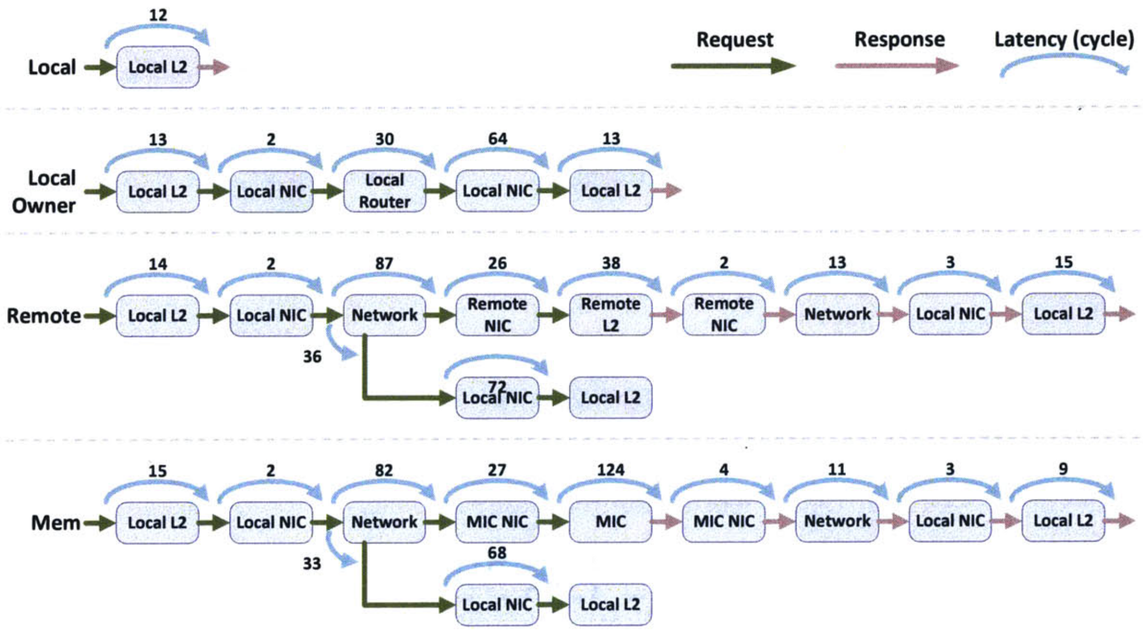


Figure 3-19: L2 Service Time Breakdown (*Barnes*)

In addition to the response, the local L2 needs to see its own request to complete the transaction which contributes to the forks in the breakdown. For both *Remote* and *Memory* requests, response travel times are faster than that of requests, as requests need to be ordered at the endpoint and cannot directly be consumed, which introduces backpressure and increases network and NIC latency, whereas the responses are unordered and can fully benefit from the low-latency network.

3.5.5 Area, Power, Timing

The SCORPIO architecture achieves distributed global ordering at low overhead. The notification network is an ultra-lightweight bufferless network with negligible area and power consumption.

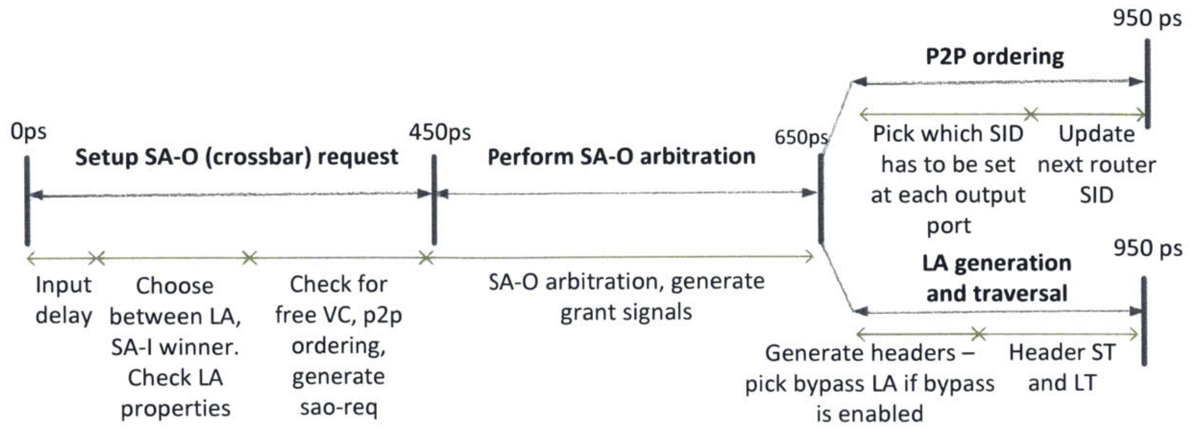
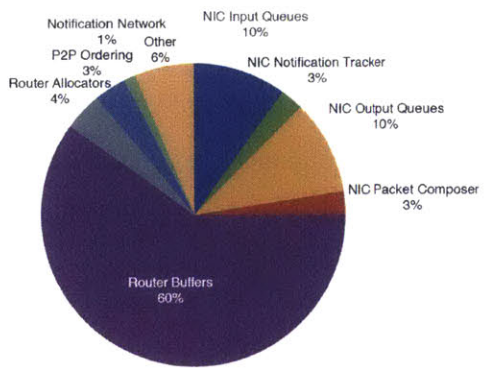

950 ps
P2P ordering
0ps
450ps
650ps
Setup SA-O (crossbar) request
Perform SA-O arbitration
Pick which SID has to be set at each output port
Update next router SID
950 ps
LA generation and traversal
Input delay
Choose between LA, SA-I winner. Check LA properties
Check for free VC, p2p ordering, generate sao-req
SA-O arbitration, generate grant signals
Generate headers – pick bypass LA if bypass is enabled
Header ST and LT

Figure 3-20: Post-synthesis Critical Path of the Main Network Router

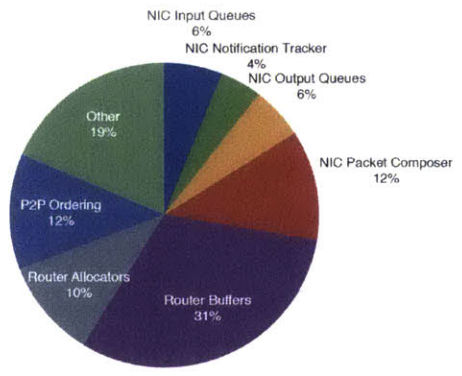
The main network is designed for high performance and the number of VCs are sized accordingly.

Timing

Figure 3-20 shows the critical path in the network, where the SA-O pipeline stage constrains the maximum clock frequency. In this stage, arbitration for output ports is performed among the winners of the SA-I stage and the lookaheads from upstream routers. Prior to this arbitration, the VC selection determines the downstream routers/directions with free space available. This influences the flits chosen for SA-O arbitration. Post-synthesis timing results show that the number of VCs does not affect the SA-O stage and is off the critical path.



(a) Area



(b) Power

Figure 3-21: Router and NIC Area and Power Breakdown

Area

The detailed area breakdown of the router and NIC is shown in Figure 3-21a, with a total area of $0.16mm^2$. The buffers in the router are the most area-hungry as they consume around 60% of the total area. While the notification network and overhead of ordering globally and point-to-point, is minimal, at 7%.

Power

The power breakdown, in Figure 3-21b, shows the power consumption of various components in the NIC and router. The average total power is around 90 mW, as it depends on the injection rate. As expected, the buffers consume a significant portion of the total power. The SID tracker consumes about 12%, and the notification tracker power overhead is negligible.

CHAPTER 4

SCORPIO: 36-CORE RESEARCH CHIP

The shared memory paradigm offers programmers a simpler model than the message passing counterpart as it allows the programmer to focus on parallelizing the application while avoiding the details of interprocessor communication. In addition data and load distribution is generally hidden from the programmer, leading to shared memory gaining widespread acceptance in multicore systems.

Shared memory multicores are prevalent in all market segments (client, server and embedded) today. Most of these existing chips use snoopy coherence (e.g. ARM ACE, Intel QPI, AMD HT), which requires ordering support such that all cores see requests in the same order. However, snoopy coherence alone cannot provide the programmers with a precise notion of shared memory semantics. The *memory consistency model* represents the memory system behavior and assumptions. In single-core processors, the memory consistency model is the sequential program order, free of ambiguity as to the validity of the read value, or result of the last write to the same memory location.

Sequential consistency, the result of extending a single-core processor memory consistency model to multicore processors, enforces (1) program order among operations on a single processors, and (2) a global sequential order among all operations. The SCORPIO network, discussed in Chapter 3 is capable of delivering globally ordered messages on a scalable packet-switched interconnect, and effectively supports snoopy coherence and sequential consistency, at low overhead. The SCORPIO 36-core chip prototype demonstrates the ease of integration of many in-order snoopy coherent cores with a scalable network that ensures global ordering. It showcases that such a

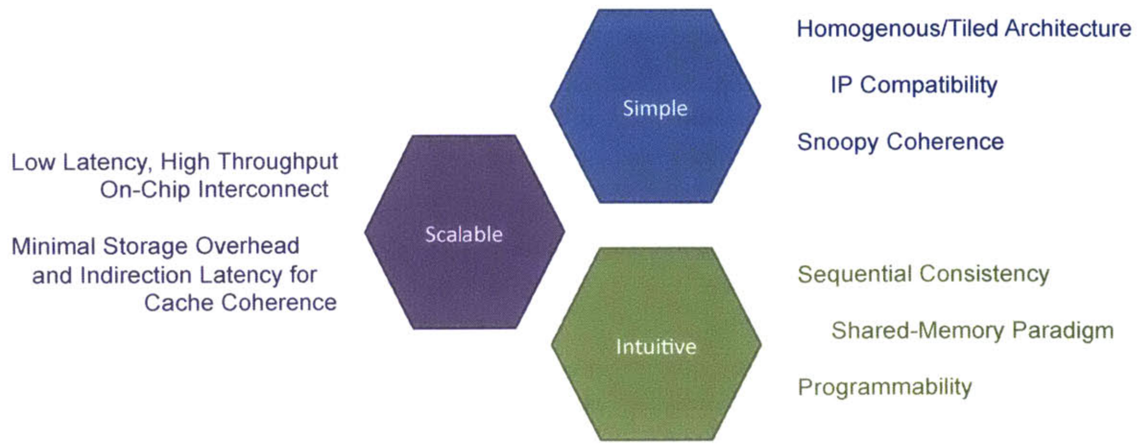

Simple
Homogenous/Tiled Architecture
IP Compatibility
Snoopy Coherence
Scalable
Low Latency, High Throughput On-Chip Interconnect
Minimal Storage Overhead and Indirection Latency for Cache Coherence
Intuitive
Sequential Consistency
Shared-Memory Paradigm
Programmability

Figure 4-1: Design Objectives of the SCORPIO Chip Prototype

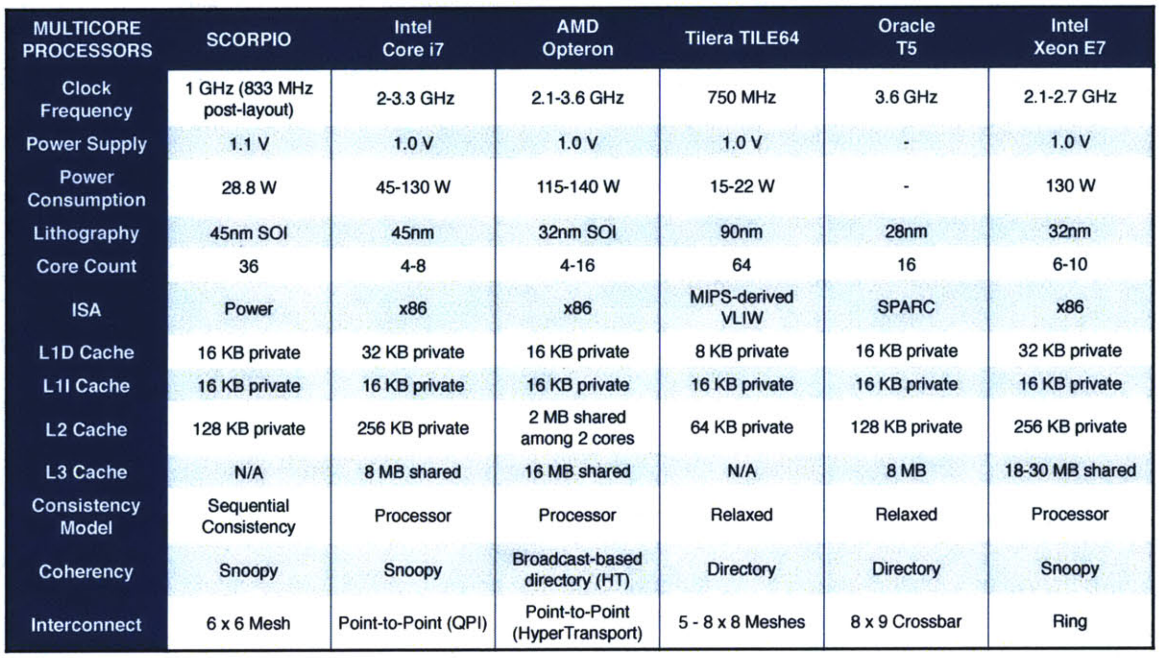
high-performance cache-coherent many-core chip can be realized at low area and power overheads.

4.1 Objectives

The SCORPIO chip must be simple, scalable, and intuitive. These driving objectives (Figure 4-1) form the basis for full system design decisions. A homogeneous/tile architecture eases layout and validation such that a large chip design and fabrication is feasible within a relatively short time. The network must be versatile such that it can readily link with many types of cores, such as the AMBA interface protocol standard which is widely accepted in commercial processors.

Snoopy coherence allows for well-designed single-core processors to be effectively connected into moderately sized multicore systems. With the low latency and high throughput on-chip ordered interconnect, snoopy coherence is readily supported on scalable mesh networks. Combined with minimal storage overhead and enhanced performance due to direct cache-to-cache transfers, we achieve simplicity, versatility, and scalability. Sequential consistency provides an intuitive view of memory to the programmer, easing programming and application development.

The novel globally ordered interconnect is showcased in an intuitive multicore processor. The effort of integrating the network with the core and creating a full memory system down to layout provides insight on the practical aspects and full system performance.



MULTICORE PROCESSORS	SCORPIO	Intel Core i7	AMD Opteron	Tilera TILE64	Oracle T5	Intel Xeon E7
Clock Frequency	1 GHz (833 MHz post-layout)	2-3.3 GHz	2.1-3.6 GHz	750 MHz	3.6 GHz	2.1-2.7 GHz
Power Supply	1.1 V	1.0 V	1.0 V	1.0 V	-	1.0 V
Power Consumption	28.8 W	45-130 W	115-140 W	15-22 W	-	130 W
Lithography	45nm SOI	45nm	32nm SOI	90nm	28nm	32nm
Core Count	36	4-8	4-16	64	16	6-10
ISA	Power	x86	x86	MIPS-derived VLIW	SPARC	x86
L1D Cache	16 KB private	32 KB private	16 KB private	8 KB private	16 KB private	32 KB private
L1I Cache	16 KB private	16 KB private	16 KB private	16 KB private	16 KB private	16 KB private
L2 Cache	128 KB private	256 KB private	2 MB shared among 2 cores	64 KB private	128 KB private	256 KB private
L3 Cache	N/A	8 MB shared	16 MB shared	N/A	8 MB	18-30 MB shared
Consistency Model	Sequential Consistency	Processor	Processor	Relaxed	Relaxed	Processor
Coherency	Snoopy	Snoopy	Broadcast-based directory (HT)	Directory	Directory	Snoopy
Interconnect	6 x 6 Mesh	Point-to-Point (QPI)	Point-to-Point (HyperTransport)	5 - 8 x 8 Meshes	8 x 9 Crossbar	Ring

Table 4.1: Comparison of Multicore Processors

4.2 Related Work

Table 4.1 includes a comparison of AMD, Intel, Tilera, SUN multiprocessors with the SCORPIO chip. These relevant efforts were a result of the continuing challenge of scaling performance while simultaneously managing frequency, area, and power. When scaling from multi to many cores, the interconnect is a significant factor. Current industry chips with relatively few cores typically use bus-based, crossbar or ring fabrics to interconnect the last-level cache, but suffers from poor scalability. Bus bandwidth saturates with more than 8 to 16 cores on-chip [24], not to mention the power overhead of signaling across a large die. Crossbars have been adopted as a higher bandwidth alternative in several multicores [3, 20], but it comes at the cost of a large area footprint that scales quadratically with core counts, worsened by layout constraints imposed by long global wires to each core. From the Oracle T5 die photo, the 8-by-9 crossbar has an estimated area of 1.5$\times$ core area, hence about $23mm^2$ at 28 nm. Rings are an alternative that supports ordering, adopted in Intel Xeon E7, with bufferless switches (called stops) at each hop delivering single-cycle latency per hop at high frequencies and low area and power. However, scaling to many cores lead to unnecessary delay when circling many hops around the die.

The Tilera TILE64 [81] is a 64-core chip with 5 packet-switched mesh networks. A successor of the MIT RAW chip which originally did not support shared memory [75], TILE64 added directory-based cache coherence, hinting at market support for shared memory. Compatibility with existing IP is not a concern for startup Tilera, with cache, directory, memory controllers developed from scratch. Details of its directory protocol are not released but news releases suggest directory cache overhead and indirection latency are tackled via trading off sharer tracking fidelity. Intel Single-chip Cloud Computer (SCC) processor [44] is a 48-core research chip with a mesh network that does not support shared memory. Each router has a four stage pipeline running at 2 GHz. In comparison, SCORPIO supports in-network ordering with a single-cycle pipeline leveraging virtual lookahead bypassing, at 1 GHz.

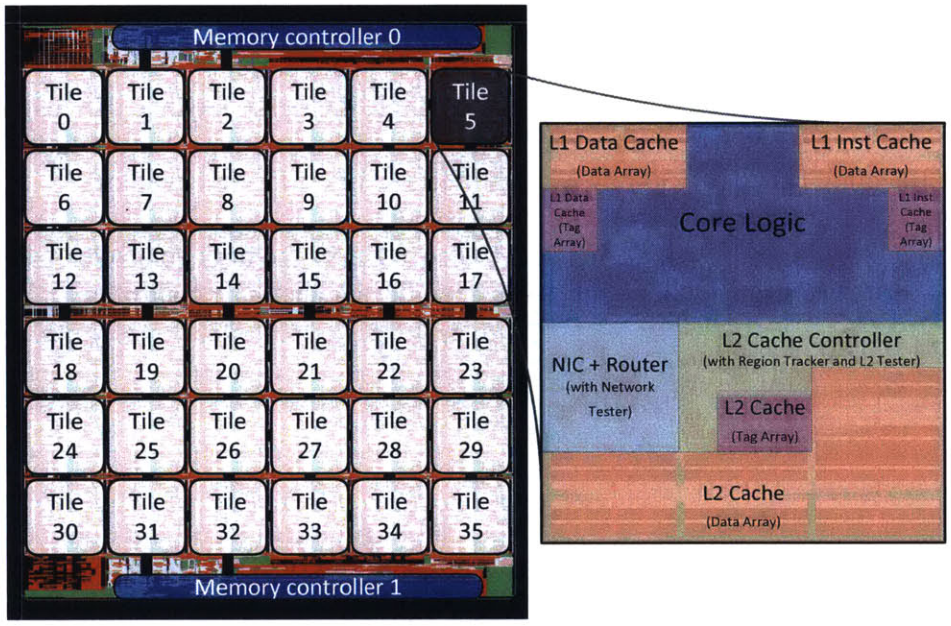


Figure 4-2: 36-Core Chip Layout and SCORPIO Tile Floorplan

4.3 Chip Overview

The 36-core fabricated multicore processor is arranged in a grid of 6$\times$6 tiles, as seen in Figure 4-2. Within each tile is an in-order core, split L1 I/D caches, private L2 cache with MOSI snoopy

Table 4.2: SCORPIO Chip Features

Process	IBM 45 nm SOI
Dimension	$11\times13\,\text{mm}^2$
Transistor count	600 M
Frequency	833 MHz
Power	28.8 W
Core	Dual-issue, in-order, 10-stage pipeline
ISA	32-bit Power Architecture$^{\text{TM}}$
L1 cache	Private split 4-way set associative write-through 16 KB I/D
L2 cache	Private inclusive 4-way set associative 128 KB
Line Size	32 B
Coherence protocol	MOSI (O: forward state)
Directory cache	128 KB (1 owner bit, 1 dirty bit)
Snoop filter	Region tracker (4 KB regions, 128 entries)
NoC Topology	6×6 mesh
Channel width	137 bits (Ctrl packets – 1 flit, data packets – 3 flits)
Virtual networks	1. Globally ordered – 4 VCs, 1 buffers each
	2. Point-to-point ordered – 2 VCs, 1 buffers each
	3. Unordered – 2 VCs, 3 buffers each
Router	XY routing, cut-through, multicast, lookahead bypassing
Pipeline	3-stage router (1-stage with bypassing), 1-stage link
Notification network	36-bits wide, bufferless, 13 cycles time window, max 4 pending messages
Memory controller	$2\times$ Dual port Cadence DDR2 memory controller + PHY
FPGA controller	$1\times$ Packet-switched flexible data-rate controller

coherence protocol, L2 region tracker for destination filtering [66], and SCORPIO NoC (see Table 4.2 for a full summary of the chip features). The commercial Power Architecture core simply assumes a bus is connected to the AMBA AHB data and instruction ports, cleanly isolating the core from the details of the network and snoopy coherence support. Between the network and the processor core IP is the L2 cache with AMBA AHB processor-side and AMBA ACE network-side interfaces. Two Cadence DDR2 memory controllers attach to four unique routers along the chip edge, with the Cadence IP complying with the AMBA AXI interface, interfacing with Cadence PHY to off-chip DIMM modules. All other IO connections go through an external FPGA board with the connectors for RS-232, Ethernet, and flash memory.

4.4 Processor Core and Cache Hierarchy

While the ordered SCORPIO NoC can plug-and-play with existing ACE coherence protocol controllers, we were unable to obtain such IP and hence designed our own. The SCORPIO cache subsystem within each tile, Figure 4-3, comprises private L1 and L2 caches, while the interaction between the self-designed L2 cache and processor core's L1 caches is subject to the core's and AHB's constraints.

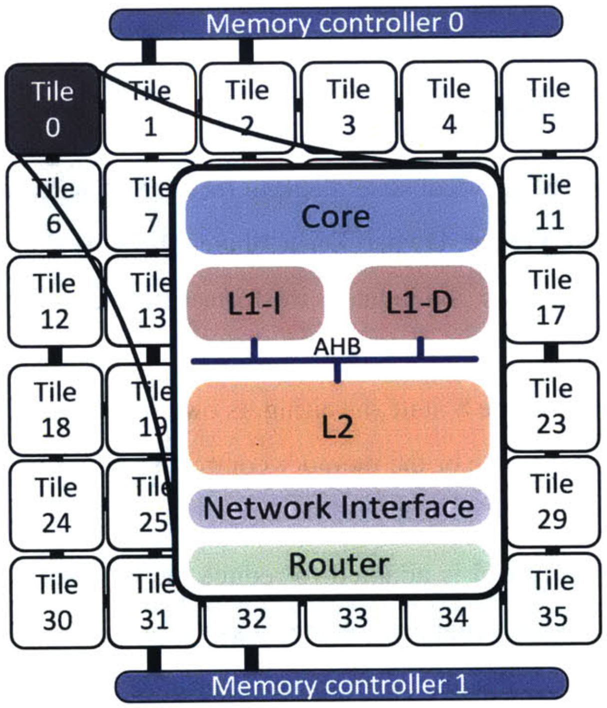


Figure 4-3: High Level Architecture of the SCORPIO Tile

The core has a split instruction and data 16 KB L1 cache with independent AHB ports. The ports connect to the multiple master split-transaction AHB bus with two AHB masters (L1 caches) and one AHB slave (L2 cache). The protocol supports a single read or write transaction at a time, hence there is a simple request or address phase, followed by a response or data phase. Transactions, between pending requests from the same AHB port, are not permitted thereby restricting the number of outstanding misses to two, one data cache miss and one instruction cache miss, per core. For multilevel caches, snooping hardware has to be present at both L1 and L2 caches. However, the core was not originally designed for hardware coherency. Thus, we added an invalidation port to the core allowing L1 cache lines to be invalidated by external input signals. This method places the inclusion requirement on the caches. With the L1 cache operating in write-through mode, the L2 cache will only need to inform the L1 during invalidations and evictions of a line.

The memory interface controller is responsible for the appropriate interfacing to the Cadence memory controller IP. It snoops requests within the network and maintains one bit per cache line, in a directory cache, to determine if the main memory is the owner of the cache line. If so, the request is sent off-chip to main memory and the data returned is sent in response to the request. This reduces the number of transaction phases if the owner is not on chip.

4.4.1 Coherence Protocol

Figure 4-4 depicts the MOSI protocol state diagram for the SCORPIO architecture. It contains the stable states (i.e, **M**odified, **O**wner, **O**wner_**D**irty, **S**hared, **I**nvalid) along with many transient states. The transient state naming convention includes the former state, desired final state, and whether it is waiting to see its own request, "A", and/or data response, "D". For instance, "IS_AD" is a transient state going from the I state to the S state, awaiting its own request and a data response. The cache line owner is either a core on-chip or the memory controller and is identified by a coherent state of either M, O, or O_D.

The standard MOSI protocol is adapted to reduce the writeback frequency and to disallow the blocking of incoming snoop requests. Writebacks cause subsequent cache line accesses to go off-chip to retrieve the data, degrading performance, hence we retain the data on-chip for as long as possible. To achieve this, an additional O_D state instead of a dirty bit per line is added to permit on-chip sharing of dirty data. For example, if another core wants to write to the same cache line, the request is broadcast to all cores resulting in invalidations, while the owner of the dirty data (in M or O_D state) will respond with the dirty data and change itself to the Invalid state. If another cores wants to read the same cache line, the request is broadcast to all cores. The owner of the dirty data (now in M state) responds with the data and transitions to the O_D state, and the requester goes to the Shared state. This ensures the data is only written to memory when an eviction occurs, without any overhead because the O_D state does not require any additional state bits.

Table 4.3 list the different processor requests, network requests, and network responses. Processor requests that miss in the cache generate network requests to obtain the necessary data or

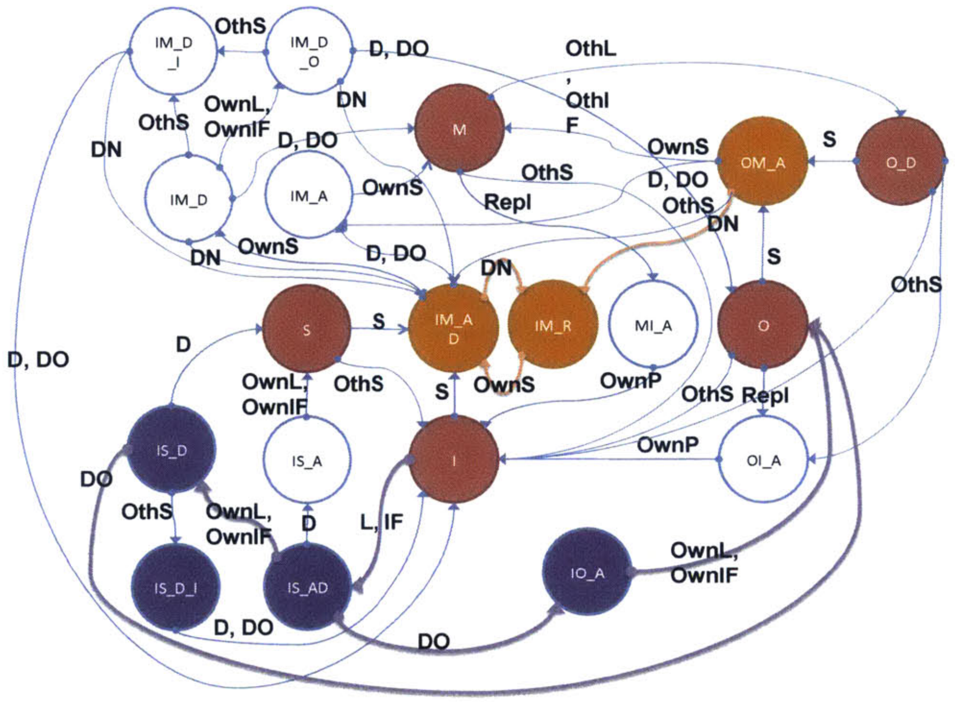


Figure 4-4: Coherence Protocol Diagram with Stable and Transient States

Table 4.3: Coherence Transaction Messages

Processor Requests	L	Load
	S	Store
	IF	Instruction Fetch
	Repl	L2 Cache Replacement
Network Requests	OwnL	Own Load Request
	OthL	Remote Load Request
	OwnIF	Own Instruction Fetch
	OthIF	Remote Instruction Fetch
	OwnS	Own Store Request
	OthS	Remote Store Request
	OwnP	Own Writeback
	OthP	Remote Writeback
Network Response	D	Data Response
	DO	Data Response with Ownership
	DN	Null Data Response

exclusive write access to the cache line. Upon the first load request to a memory location, the cache is not populated and all entries are marked *Invalid*. The request is serviced by main memory, which responds with either a D or DO response. Load requests from many nodes and all to the same

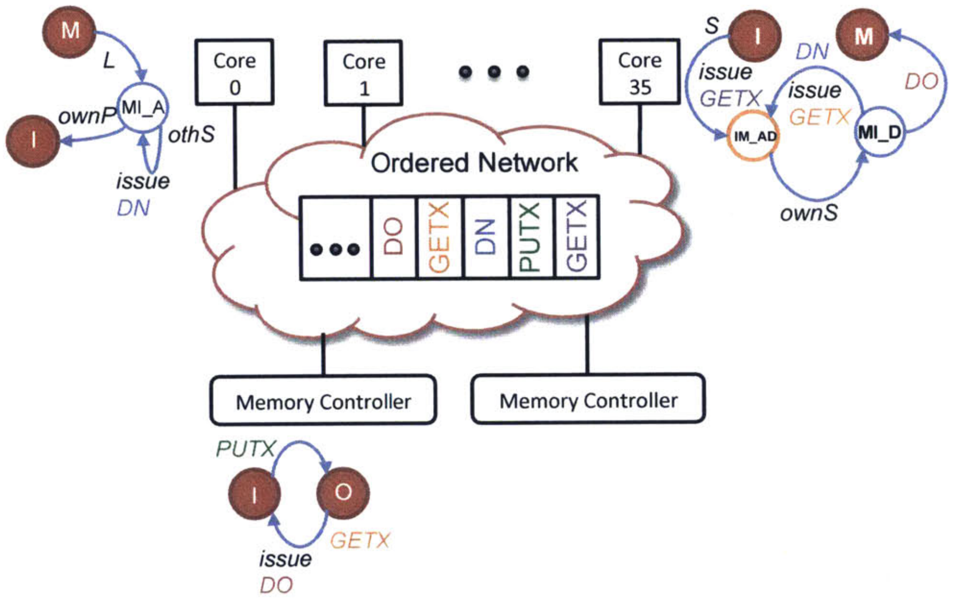


Figure 4-5: Writeback Case with GetX Retry

memory location will result in each going to main memory to obtain the data, even though the data is already cached on-chip. The DO network response indicates that the main memory is passing ownership to the requester. Future accesses to that cache line will be serviced by the on-chip cache with ownership. The purple shaded transient states in Figure 4-4 identify the cache state transitions from *Invalid* to *Owner*.

Writeback Race Handling

Writebacks need to be ordered, but to avoid broadcasting multi-flit writeback data, we implemented writeback as a request broadcast on GO_REQ along with a data unicast on UO_RESP to the memory controller. The memory controller handles the writeback as two packets and keeps track of the request and data in any order it is received. When the memory controller receives the data packet, it logs in the directory cache that the data was received. The last to arrive of the two writeback packets will be pushed to off-chip memory. While the writeback is pending, other requests and responses can be continually serviced provided there are no data dependencies.

This creates a potential race, when a writeback is injected into the network, possibly due to a cache line replacement, and another core performs a store miss and issues a request for ownership (GETX) into the network. This example is depicted in Figure 4-5. Core 0 has a dirty cache line that is evicted and generates a PUTX request. In global order though, the GETX from core 35 arrives before the ownP request. Responding to the GETX would be incorrect because the data and PUTX request cannot be cancelled at the memory controller. This results in two owners of the cache line. Thus, we introduce the GETX retry mechanism where the core doing the store miss, Core 35, will receive null data. Upon reception of the null data, it will re-inject the GETX request into the network. The coherent state transitions can be seen in Figure 4-5, and the additional states for handling the GETX retry are shown. Since the null data response is necessary for the protocol function, the network's data response virtual network has to handle varying data packet sizes.

Data Forwarding

When a cache line is in a transient state due to a pending write request, snoop requests to the same cache line are stalled until the data is received and the write request is completed. This causes the blocking of other snoop requests even if they can be serviced right away. We service all snoop requests without blocking by maintaining a forwarding IDs (FID) list that tracks subsequent snoop requests that match a pending write request. The FID consists of the SID and the request entry ID or the ID that matches a response to an outstanding request at the source. With this information, a completed write request can send updated data to all SIDs on the list. The core IP has a maximum of 2 outstanding messages at a time, hence only two sets of forwarding IDs are maintained per core. The SIDs are tracked using a N bit-vector, and the request entry IDs are maintained using 2N bits. For larger core counts and more outstanding messages, this overhead can be reduced by tracking a smaller subset of the total core count. Since the number of sharers of a line is usually low, this will perform as well as being able to track all cores. Once the FID list fills up, subsequent snoop requests will then be stalled.

ACE Interface

The different messages types are matched with appropriate ACE channels and types. The network interface retains its general mapping from ACE messages to packet type encoding and virtual network identification resulting in a seamless integration. The L2 cache was thus designed to comply with the AMBA ACE specification. It has five outgoing channels and three incoming channels (see Figure 4-6), separating the address and data among different channels. ACE is able

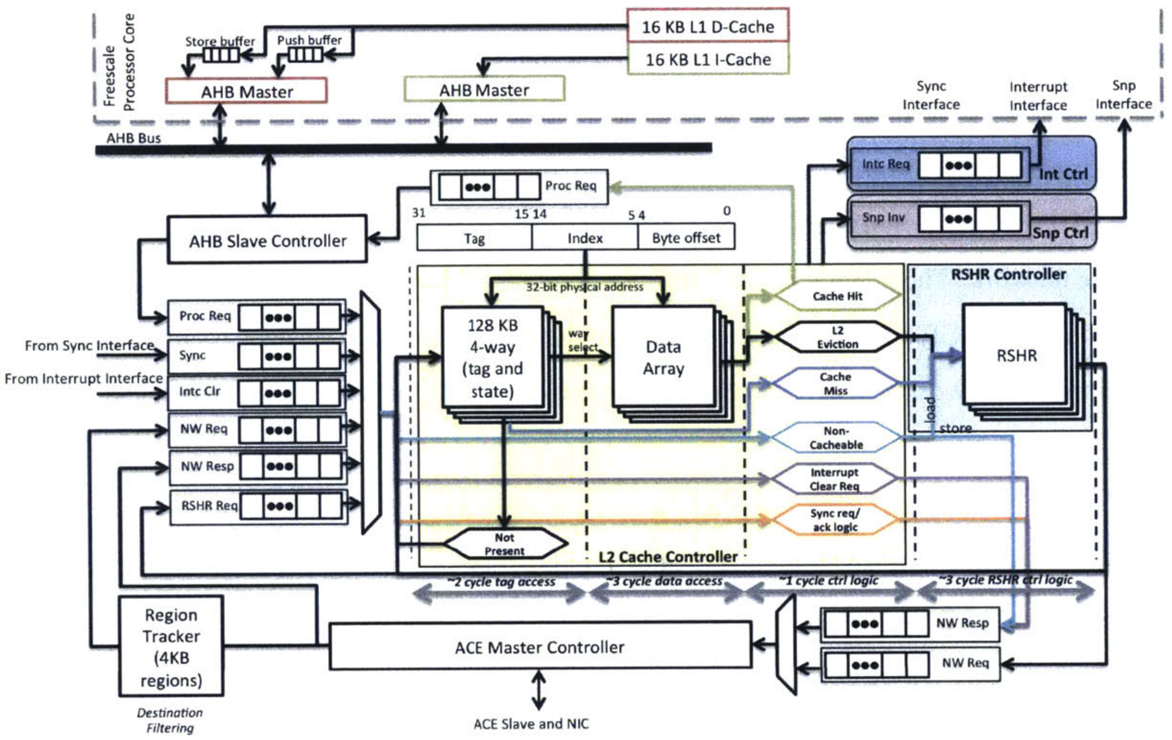


Figure 4-6: L2 Cache and NIC Microarchitecture

to support snoop requests through its Address Coherent (AC) channel, allowing us to send other requests to the L2 cache.

4.4.2 L2 Microarchitecture

The L2 microarchitecture, depicted in Figure 4-6, contains the AHB slave for core-side interface, ACE master for network-side interface, L2 cache, request status holding register (RSHR), interrupt, and sync controllers. SCORPIO's L2 cache is not pipelined for simplicity, since the number of maximum outstanding L1 requests is small and we expect the number of snoops to be low with high hit rates and snoop filtering.

When a L1 data read miss occurs, the request is buffered in the processor request FIFO until it wins arbitration to the tag arrays. The 17-bit tag lookup is performed across four ways associated with the 10-bit index. The tag and data arrays utilize the IBM 45nm SOI SRAMs that operate slower than the 1 GHz global clock requiring multiple read and write cycles. Depending on the tag comparison and state, there are alternative paths through the multi-stage L2 controller, such as cache hit, cache miss, not present and eviction. The cache miss path in Figure 4-6 bypasses the data array access and heads to the RSHR where an entry is allocated followed by the injection of a network load request into the outgoing network request (NW Req) queue. The response returns with the matching RSHR entry ID so an entire 32-bit address does not need to be sent with the data, which reduces the data packet size. The data and coherent state are then updated, the RSHR entry deallocated, and the data response sent to the processor core. Essentially, the L2 controller itself handles the stable states while the RSHR controller deals with the transient states.

The RSHR contains four entries used to store the pending requests awaiting a network response. The scenario when all entries are occupied occurs when (1) instruction fetch miss in L1 and L2 - allocate line and (2) evict an existing cache line in Owned or Modified state, (3) data cache miss in L1 and L2 occurs due to independent AHB master from the instruction cache, (4) allocate a line and evict an existing cache line in Owned and Modified state.

Synchronization with *msync*

The Power Architecture core enforces program-order, essentially meaning all memory accesses appear to be executed in program order. The internal core architecture has a single pipeline to the cache and MMU, ensuring this ordering. But, in a multiprocessor system, the in-order core actually follows a weak consistency model. The core contains store buffers that hold write-through requests queued for AHB access without stalling the pipeline. For synchronization and barriers, the core uses the *msync* instruction. Its purpose is to flush any pending stores of the executing processor and service pending snoops across all processors.

A RSHR entry is reserved specifically for msync requests. The evicted lines are retained in the RSHR until the ownP request is seen in global order from the network. SCORPIO supports global ordering within the network, allowing all cores to view the same program operations in the same order despite any inter-processor interleaving that occurs. Hence, to support the *msync* instruction, we inject it into SCORPIO's ordered network GO-REQ. Upon receiving the sync request, each core empties the snoop queue and issues an acknowledgement. This ACK is sent on SCORPIO's UO-RESP network to the *msync* requester, where it is counted with the other cores' ACKs. When N-1 ACKs are received, where N is the number of cores, the requester asserts a 1-bit active-high input signal to the processor's sync interface indicating the completion of the *msync* instruction. If an interrupt occurs during a pending `msync` operation, the synchronization is aborted and restarted at a later time.

Mutual Exclusion

For a shared memory system, atomic operations are essential for mutual exclusion of cache lines. In the PowerPC instruction set, there are LWARX and STWCX instructions that opportunistically attempt to lock and unlock a cache line. The load lock or LWARX can be completed locally by many cores but only one store unlock or STWCX will succeed. The coherence protocol is not significantly affected by the atomic operations. The lock bit is set when the own load request is seen in global order. Similarly, the lock bit is cleared if another core's store request is seen prior to core's own STWCX instruction. Locks introduce more complexity and communication between the independent controllers within the L2 cache. Within the RSHR, the lock instruction is indicated by a 1-bit identifying it as a lock operation or not. If it is a LWARX, the lock bit is set upon allocation of a RSHR entry. Upon reception of the ownL request, the RSHR controller informs the L2 cache controller to update the lock bit in the tag array. As all write operations through the processor core are of double-word widths, while the network supports cache line size transfers, we merge multiple write operations to save on bandwidth. This mechanism is straightforward, but for STWCX, care must be taken to preserve ordering semantics. Thus, STWCX's 64-bit write data cannot be merged with the cache line data response till the STWCX's ownS request arrives with its lock bit still asserted. At which point, the data will be merged and stored in the L2 data array.

Non-Cacheable Reads/Writes

These are double-word in size and bypass the coherent state logic. For non-cacheable reads, the RSHR is still used to await the data response from the memory controller. For writes, record keeping is not necessary because the write's injection into the outgoing queues signals completion of that operation. If the L2 cache is disabled, all non-atomic coherent transactions are converted to non-cacheable requests. The non-cacheable request will be sent straight to the memory controller and are not broadcasted to the other cores.

MMU Interrupts

The software-managed MMU in the processor core interacts with the TLB through MMU assist registers. In a multiprocessor system, synchronization of TLB operations is required between the linux kernel on all processors. The core does not support TLB synchronization instructions and inter-processor interrupts are used to broadcast the TLB invalidations. When the issuing core accesses the memory-mapped interrupt address then the network broadcasts to all the cores. Other snooping L2 caches, receive this interrupt and propagates it to the core through external interrupt signals. When the MIC receives the interrupt, it begins the interrupt controller state machine by initializing N flag bits, where N is the number of cores. A 1-bit ack signal is generated when each core completes the interrupt routine, sending a clear flag request with the SID to the memory controller. When all flag bits are cleared, then all the cores may proceed.

Algorithm 1 Sync regression test for 2 cores

```
1:  if $core\_id = 0$ then
2:      $A \leftarrow 1$
3:      asm volatile ("sync" : : : "memory")        ▷ A's value seen by all other cores
4:      $B \leftarrow 1$
5:      asm volatile ("sync" : : : "memory")        ▷ B's value seen by all other cores
6:  else
7:      while $B = 0$ do
8:      end while
9:      if $A \neq 0$ then
10:         $exit\_failed()$
11:     end if
12: end if
13: $exit\_succeeded()$
```

4.5 Functional Verification

We ensure correct functionality of the SCORPIO RTL using a suite of regression tests, listed in Table 4.4, that verify the entire chip. Since the core is verified commercial IP, our regression tests focus on verifying integration of various components, which involves (1) load/store operations on both cacheable and non-cacheable regions, (2) lock and barrier instructions, (3) coherency between L1s, L2s and main memory, and (4) software-triggered interrupts. For brevity, Algorithm 1 shows the code segment of the shortest *sync* test. The tests are written in assembly and C, and a custom-built software chain compiles tests into machine code.

Table 4.4: Regression Tests

Test name	Description
hello	Performs basic load/store and arithmetic operations on non-overlapped cacheable regions.
mem patterns	Performs load/store operations for different data types on non-overlapped cacheable regions.
config space	Performs load/store operations on non-cacheable regions.
flash copy	Transfer data from the flash memory to the main memory.
sync	Uses flags and performs msync operation.
atom smashers	Uses spin locks, ticket locks and ticket barriers and performs operations on shared data structures.
ctt	Performs a mixture of arithmetic, lock, load/store operations on overlapped cacheable regions.
intc	Performs store operations on the designate interrupt address which triggers other cores' interrupt handler.

4.6 Architecture Analysis

Modeled system. For full-system architectural simulations of SCORPIO, we use Wind River Simics [5] extended with the GEMS toolset [59] and the GARNET [10] network model. The SCORPIO and baseline architectural parameters as shown in Table 4.2 are faithfully mimicked within the limits of the GEMS and GARNET environment:

- GEMS only models in-order SPARC cores, instead of SCORPIO's Power cores.
- L1 and L2 cache latency in GEMS are fixed at 1 cycle and 10 cycles. The prototype L2 cache latency varies with request type and cannot be expressed in GEMS, while the L1 cache latency of the core IP is 2 cycles.
- The directory cache access latency is set to 10 cycles and DRAM to 80 cycles in GEMS. The off-chip access latency of our chip prototype is unknown as it depends on the PCB board and packaging, which is still being designed. The directory cache access was approximated from the directory cache parameters, but will also vary depending on request type for the chip.

- The L2 cache, NIC, and directory cache accesses are fully-pipelined in GEMS.
- Maximum of 16 outstanding messages per core in GEMS, unlike our chip prototype which has a maximum of two outstanding messages per core.

Directory baselines. For directory coherence, all requests are sent as unicasts to a directory, which forwards them to the sharers or reads from main memory if no sharer exists. SCORPIO is compared with two baseline directory protocols. The *Limited-pointer directory* (LPD) [9] baseline tracks when a block is being shared between a small number of processors, using specific pointers. Each directory entry contains 2 state bits, log N bits to record the owner ID, and a set of pointers to track the sharers. We evaluated LPD against full-bit directory in GEMS 36 core full-system simulations and discovered almost identical performance when approximately 3 to 4 sharers were tracked per line as well as the owner ID. Thus, the pointer vector width is chosen to be 24 and 54 bits for 36 and 64 cores, respectively. By tracking fewer sharers, more cache lines are stored within the same directory cache space, resulting in a reduction of directory cache misses. If the number of sharers exceeds the number of pointers in the directory entry, the request is broadcast to all cores. The other baseline is derived from *HyperTransport* (HT) [23]. In HT, the directory does not record sharer information but rather serves as an ordering point and broadcasts the received requests. As a result, HT does not suffer from high directory storage overhead but still incurs on-chip indirection via the directory. Hence for the analysis only 2 bits (ownership and valid) are necessary. The ownership bit indicates if the main memory has the ownership; that is, none of the L2 caches own the requested line and the data should be read from main memory. The valid bit is used to indicate whether main memory has received the writeback data. This is a property of the network, where the writeback request and data may arrive separately and in any order because they are sent on different virtual networks.

Workloads. We evaluate all configurations with SPLASH-2 [4] and PARSEC [14] benchmarks. Simulating higher than 64 cores in GEMS requires the use of trace-based simulations, which fail to capture dependencies or stalls between instructions, and spinning or busy waiting behavior accurately. Thus, to evaluate SCORPIO's performance scaling to 100 cores, we obtain SPLASH-2 and PARSEC traces from the Graphite [63] simulator and inject them into the SCORPIO RTL.

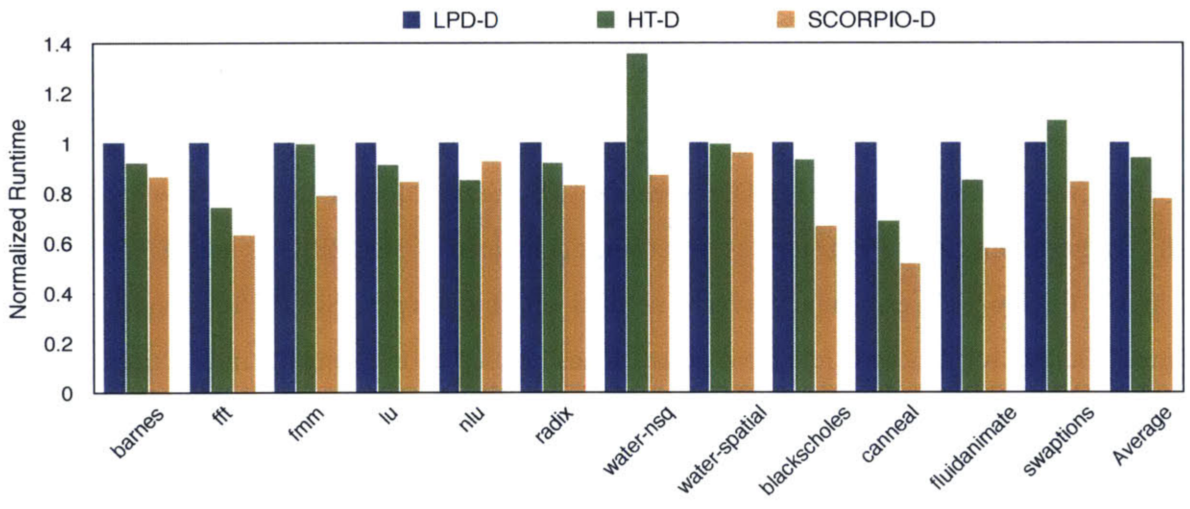
Evaluation Methodology. For performance comparisons with baseline directory protocols and prior in-network coherence proposals, we use GEMS to see the relative runtime improvement. The centralized directory in HT and LPD adds serialization delay at the single directory. Multiple distributed directories alleviates this but adds on-die network latency between the directories and DDR controllers at the edge of the chip for off-chip memory access, for both baselines. We evaluate the distributed versions of LPD (LPD-D), HT (HT-D), and SCORPIO (SCORPIO-D) to equalize this latency and specifically isolate the effects of indirection and storage overhead. The directory cache is split across all cores, while keeping the total directory size fixed to 256 KB. Our chip prototype uses 128KB, as seen in Table 4.2, but we changed this value for baseline performance comparisons only so that we do not heavily penalize LPD by choosing a smaller directory cache.

The SCORPIO network design exploration provides insight into the performance impact as certain parameters are varied. The finalized settings from GEMS simulations are used in the fabricated 36-core chip NoC. In addition, we use behavioral RTL simulations on the 36-core SCORPIO RTL, as well as 64 and 100-core variants, to explore the scaling of the uncore to high core counts. For reasonable simulation time, we replace the Cadence memory controller IP with a functional memory model with fully-pipelined 90-cycle latency. Each core is replaced with a memory trace injector that feeds SPLASH-2 and PARSEC benchmark traces into the L2 cache controller's AHB interface. We run the trace-driven simulations for 400 K cycles, omitting the first 20 K cycles for cache warm-up.

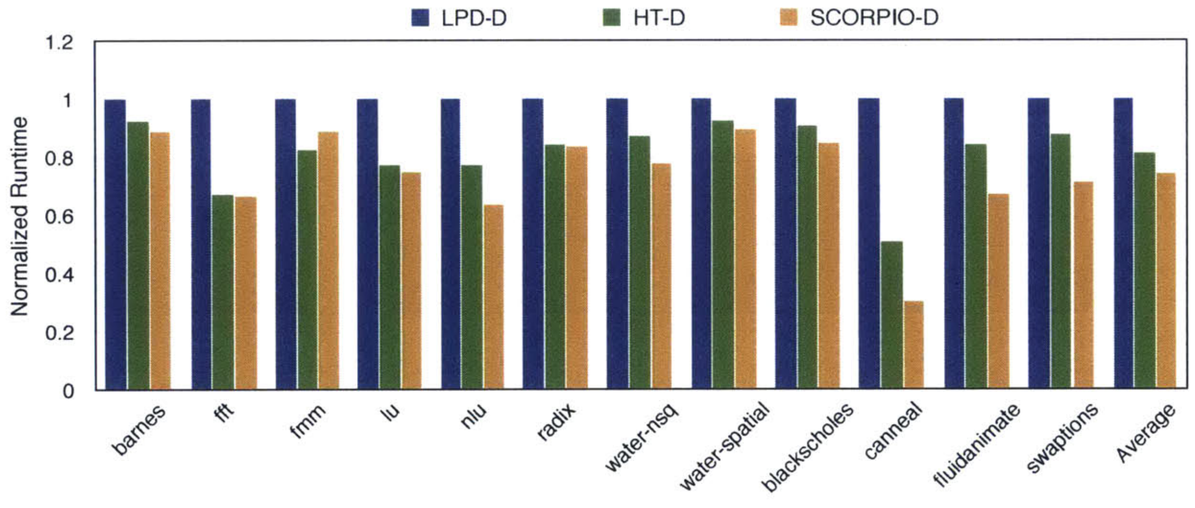
We evaluate the area and power overheads to identify the practicality of the SCORPIO NoC. The area breakdown is obtained from layout. For the power consumption, we perform gate-level simulation on the post-synthesis netlist and use the generated value change dump (VCD) files and Synopsys PrimeTime PX. To reduce the simulation time, we use trace-driven simulations to obtain the L2 and network power consumption. We attach a mimicked AHB slave, that responds to memory requests in a few cycles, to the core and run the Dhrystone benchmark to obtain the core power consumption.

4.6.1 Performance

To ensure the effects of indirection and directory storage are captured in the analysis, we keep all other conditions equal. Specifically, all architectures share the same coherence protocol and run on



(a) 36 Cores



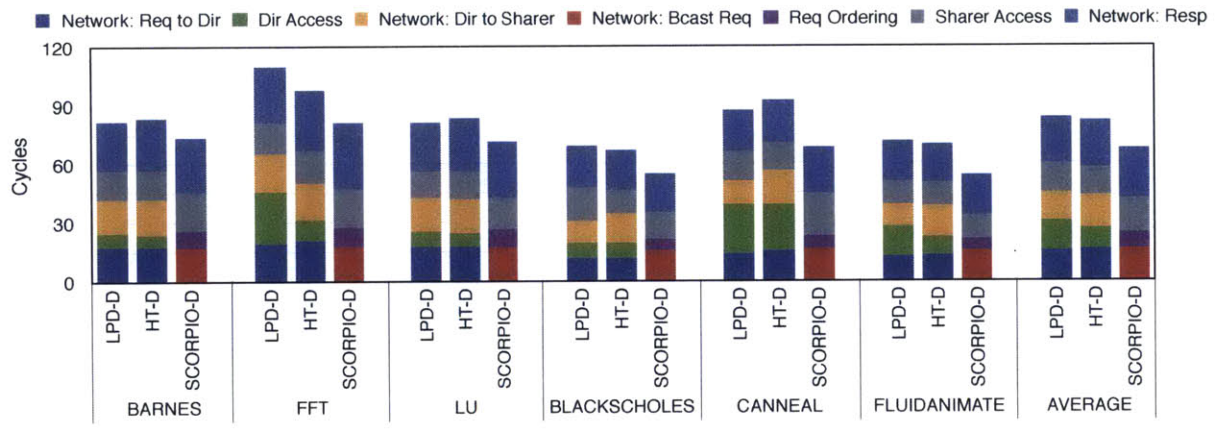
(b) 64 Cores

Figure 4-7: Application Runtime for 36 and 64 Cores

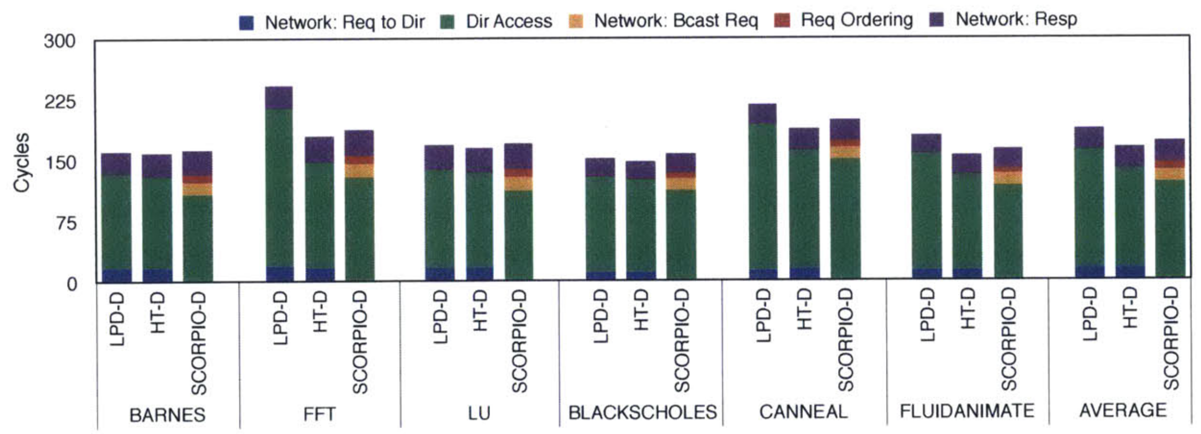
the same NoC (minus the ordered virtual network GO-REQ and notification network).

Figure 4-7 shows the normalized full-system application runtime for SPLASH-2 and PARSEC benchmarks simulated on GEMS. On average, SCORPIO-D shows 24.1% better performance over LPD-D and 12.9% over HT-D across all benchmarks. Diving in, we realize that SCORPIO-D experiences average L2 service latency of 78 cycles, which is lower than that of LPD-D (94 cycles) and HT-D (91 cycles). The average L2 service latency is computed over all L2 hit, L2 miss (including off-chip memory access) latencies and it also captures the internal queuing latency between the core and the L2. Since the L2 hit latency and the response latency from other caches or memory controllers are the same across all three configurations, we further breakdown request delivery latency for three SPLASH-2 and three PARSEC benchmarks (see Figure 4-8). When a request is served by other caches, SCORPIO-D's average latency is 67 cycles, which is 19.4% and 18.3% lower than LPD-D and HT-D, respectively. Since we equalize the directory cache size for all configurations, the LPD-D caches fewer lines compared to SCORPIO-D and HT-D, leading to a higher directory access latency which includes off-chip latency. SCORPIO provides the most latency benefit for data transfers from other caches on-chip by avoiding the indirection latency.

As for requests served by the directory, HT-D performs better than LPD-D due to the lower directory cache miss rate. Also, because the directory protocols need not forward the requests to other caches and can directly serve received requests, the ordering latency overhead makes the



(a) Served by other caches



(b) Served by directory

Figure 4-8: Application Latency Breakdown for 36 Cores

SCORPIO delivery latency slightly higher than the HT-D protocol. Since the directory only serves 10% of the requests, SCORPIO still shows 17% and 14% improvement in average request delivery latency over LPD-D and HT-D, respectively, leading to the overall runtime improvement.

4.6.2 Design Space Exploration

We perform GEMS simulations to view the performance impact of SPLASH-2 and PARSEC applications as the L2 cache and directory sizes are changed. Combined with the NoC design exploration in Chapter 3, we identify the finalized hardware to be fabricated.

L2 Sizing

L2 cache sizing has a tremendous impact on the full system performance and we sized SCORPIO's L2 to yield moderate performance while ensuring high core counts within a limited die area. Our design choices are arrived through design space explorations using GEMS full system performance simulations, while considering design overheads. The simulation consists of 32 of the 36 cores activated and operating on the benchmark programs. The programs run on a power-of-two number of cores and is deemed to be sufficient for design space exploration. It can be assumed that the other 4 cores are idle.

The graph in Figure 4-9 is the normalized runtime for a sweep of the L2 cache size parameter. The primary goal for increasing the number of cores on the limited chip area comes at the cost of L2 cache size. The results show that for 128KB L2 caches, application runtime improves by approximately 40% over that with 64KB L2 caches. A smaller cache size of 64KB will not be

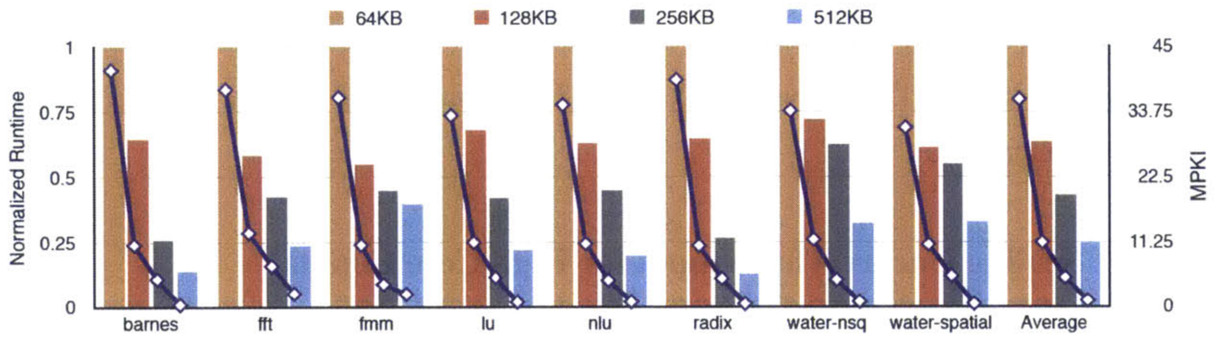


Figure 4-9: Application Performance with Varying L2 Cache Sizes

suitable for overall performance since SCORPIO's L2 has to be inclusive of the 32KB L1. A 256KB cache size can further improve runtime by 20%, but leads to 30% tile area bloat which reduces the core count to 27, within the same 11x13mm die area, based on post-layout area numbers. In addition, when we analyze the number of L2 misses per thousand instructions , we saw that on-chip traffic can be too low for these benchmarks to be able to see the impact of SCORPIO's NoC.

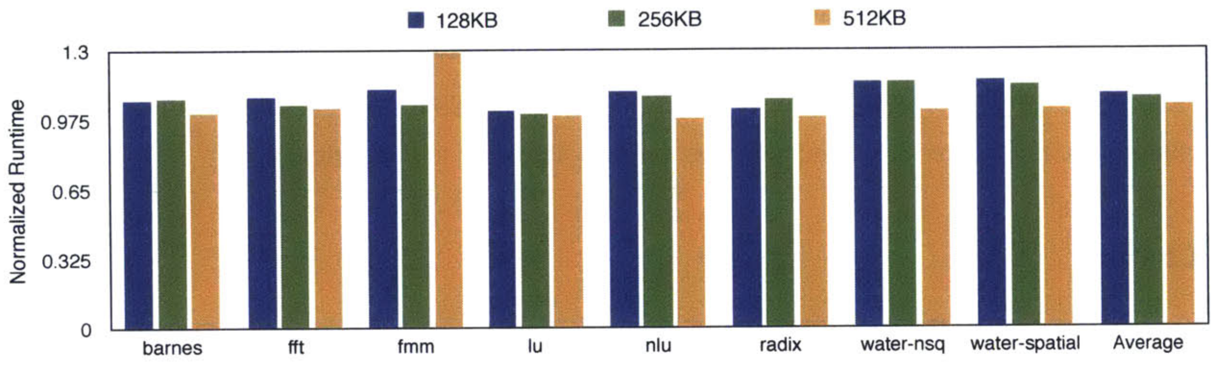


Figure 4-10: Application Performance with Varying Directory Cache Sizes

Directory Sizing

A small directory at the memory controller frees up the controller for requests that indeed need to be serviced by main memory. Thus, as the memory controller snoops requests, it determines if the owner of the cache line is on-chip and whether to respond to the request. For applications that access many cache lines simultaneously, the directory cache may fall short of the size required to hold the status of each cache line. If an entry is not found, the request is sent to the memory controller, even if an on-chip owner exists. Thus, both will respond with the data. We sweep the directory size to determine its impact on the application performance, and finalize the size for the chip fabrication. Figure 4-10 shows only a slight increase in runtime as the size is varied. We size the directory cache to 128 kB, on the order of a L2 cache size, at the memory controllers, to hold the owner information.

4.6.3 Scaling Uncore Throughput for High Core Counts

As core counts scale, if each core's injection rate (cache miss rate) remains constant, the overall throughput demand on the uncore scales up. We explore the effects of two techniques to optimize

SCORPIO's throughput for higher core counts.

Pipelining the Uncore

Pipelining the L2 caches improves its throughput and reduces the backpressure on the network which may stop the NIC from de-queueing packets. Similarly, pipelining the NIC will relieve network congestion. The performance impact of pipelining the L2 and NIC can be seen in Figure 4-11 in comparison to a non-pipelined version. For 36 and 64 cores, pipelining reduces the average latency by 15% and 19%, respectively. Its impact is more pronounced as we increase to 100 cores, with an improvement of 30.4%.

Boosting Main Network Throughput with Virtual Channels.

For good scalability on any multiprocessor system, the cache hierarchy and network should be co-designed. As core count increases, assuming similar cache miss rates and thus traffic injection rates, the load on the network now increases. The theoretical throughput of a $k \times k$ mesh is $1/k^2$ for broadcasts, reducing from 0.027 flits/node/cycle for 36-cores to 0.01 flits/node/cycle for 100-cores. Even if overall traffic across the entire chip remains constant, say due to less sharing or larger caches, a 100-node mesh will lead to longer latencies than a 36-node mesh. Common ways to boost a mesh throughput include multiple meshes, more VCs/buffers per mesh, or wider channel.

Within the limits of the RTL design, we analyze the scalability of the SCORPIO architecture by varying core count and number of VCs within the network and NIC, while keeping the injection rate constant. The design exploration results show that increasing the UO-RESP virtual channels

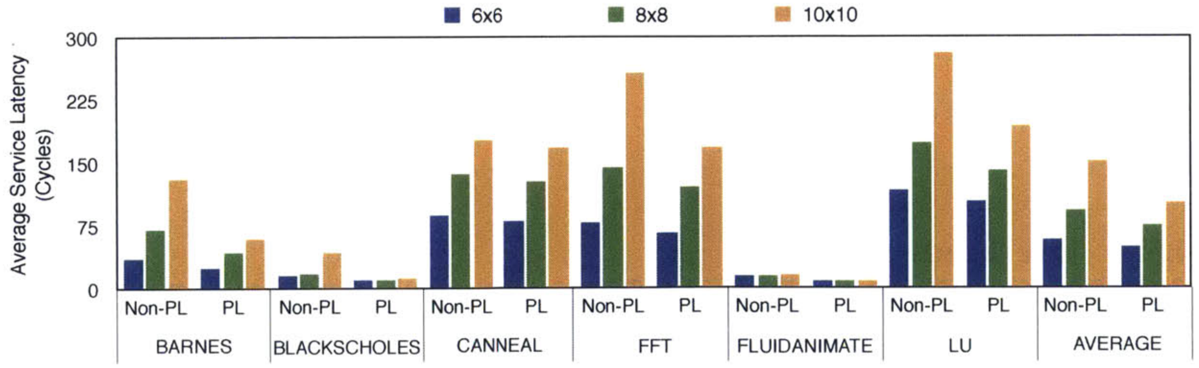


Figure 4-11: Pipelining Effect on Performance and Scalability

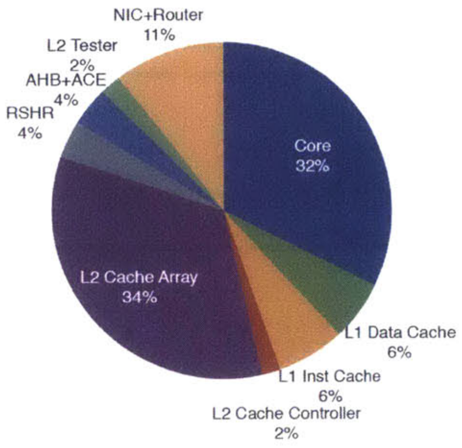
does not yield much performance benefit. But, the OREQ virtual channels matter since they support the broadcast coherent requests. Thus, we increase only the OREQ VCs from 4 VCs to 16 VCs (64 cores) and 50 VCs (100 cores), with 1 buffer per VC. Further increasing the VCs will stretch the critical path and affect the operating frequency of the chip. It will also affect area, though with the current NIC+router taking up just 10% of tile area, this may not be critical. A much lower overhead solution for boosting throughput is to go with multiple main networks, which will double/triple the throughput with no impact on frequency. It is also more efficient area wise as excess wiring is available on-die.

For at least 64 cores in GEMS full-system simulations, SCORPIO performs better than LPD and HT despite the broadcast overhead. The 100-core RTL trace-driven simulation results in Figure 4-11 show that the average network latency increases significantly. Diving in, we realize that the network is very congested due to injection rates close to saturation throughput. Increasing the number of VCs helps push throughput closer to the theoretical, but is ultimately still constrained by the theoretical bandwidth limit of the topology. A possible solution is to use multiple main networks, which would not affect the correctness because we decouple message delivery from ordering. Our trace-driven methodology could have a factor on the results too, as we were only able to run 20K cycles for warmup to ensure tractable RTL simulation time; we noticed that L2 caches are under-utilized during the entire RTL simulation runtime, implying caches are not warmed up, resulting in higher than average miss rates.

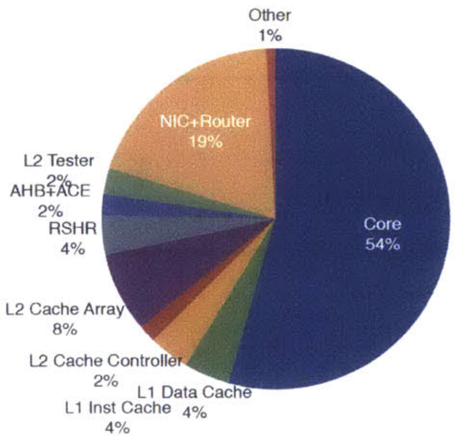
An alternative to boosting throughput is to reduce the bandwidth demand. INCF [11] was proposed to filter redundant snoop requests by embedding small coherence filters within routers in the network. This is left for future work.

4.6.4 Overheads

In Chapter 3, the SCORPIO NoC area and power breakdown is discussed. Here we evaluate the portion of the entire tile area and power consumed by the SCORPIO NoC.



(a) Area Breakdown



(b) Power Breakdown

Figure 4-12: SCORPIO Tile Area and Power

Area

The dimension of the fabricated SCORPIO is $11 \times 13\,mm^2$. Each memory controller and each memory interface controller occupies around $5.7\,mm^2$ and $0.5\,mm^2$ respectively. Detailed area breakdown of a tile is shown in Figure 4-12b. Within a tile, L1 and L2 caches are the major area contributors, taking 46% of the tile area and the network interface controller together with router occupying 10% of the tile area.

Power

Overall, the aggregated power consumption of SCORPIO is around 28.8 W and the detailed power breakdown of a tile is shown in Figure 4-12a. The power consumption of a core with L1 caches is around 62% of the tile power, whereas the L2 cache consumes 18% and the NIC and router 19% of tile power. A notification router costs only a few OR gates; as a result, it consumes less than 1% of the tile power. Since most of the power is consumed by clocking the pipeline and state-keeping flip-flops for all components, the breakdown is not sensitive to workload.

CHAPTER 5

SCEPTER: HIGH-PERFORMANCE BUFFERLESS NOC ARCHITECTURE

Over the last few decades, the power wall has led to a shift from uniprocessors to multicore processors, and to the point where the interconnect plays a large role in achieving the desired performance of multicore chips. Traditional networks such as buses and crossbars are displaced by network-on-chip (NoC) architectures to achieve high-bandwidth and scalability. The packet-switched NoC architectures must be capable of delivering high bandwidth at low latencies and still remain within a tight power budget. However, the network consumes a significant portion of total chip power: The MIT RAW [76] chip network, which connects 16 tiles, consumes about 40% of the tile power; The Intel TeraFLOPS [43] chip network, that connects 80 tiles, and consumes 30% of the total power ; The MIT SCORPIO [28] NoC, connects 36 tiles, and consumes 18% of the tile power.

5.1 Motivation

The SCORPIO chip's buffers consume a significant portion of the tile power. With L2 cache power consuming 19% of the tile power, the network power consumption on par with the L2 cache is disconcerting. An area overhead of 10% is not alarming, however for improved performance the number of VCs must be increased substantially leading to significant network area consumption.

Bufferless NoCs [30,48,65] have been proposed as an alternative to virtual channel buffered NoCs, tackling the high network power head-on by eliminating buffers. At each router, the flits that enter must exit the router as there are no buffers. If multiple flits contend for an output port, one succeeds and the rest are deflected to other output directions. Although the elimination of buffers results in 40% [65] power savings, the operating point is limited to low and medium network loads as performance degrades quickly for higher loads. The two key contributors to the performance wall encountered in bufferless NoCs, are (1) the network latency increases as a result of contention/deflections in the network and is proportional to the number of hops and deflection rate, and (2) congestion exacerbates the contention and results in quick throughput degradation and starvation.

To achieve high performance the network latency must be reduced to effectively drain the network of flits efficiently and the congestion needs to be controlled for high network loads to achieve reduced starvation.

5.2 Related Work

5.2.1 Bufferless NoCs and Congestion Control

A baseline bufferless NoC was described in Section 2.2. Different NoC architectures have been recently proposed to address the challenges and power/performance optimization. Although there is a large literature space for congestion control and bufferless network performance, a few relevant works are CHIPPER [30], MinBD [31], Clumsy Flow Control (CFC) [48], self-tuned congestion control [77] and Heterogeneous Adaptive Throttling (HAT) [18]. The CHIPPER bufferless NoC reduces the complexity of the router by replacing the expensive port allocation and crossbar with a permutation network. It also proposes the use of cache miss buffers for re-assembly of multi-flit packets and a retransmit protocol when endpoint buffers are full. CHIPPER reduces the critical path of the router, but saturates more quickly than BLESS and buffered networks. BLESS is a good baseline for bufferless NoCs as its performs either equal to or better than CHIPPER for all workloads evaluated in [30].

HAT and CFC utilize source throttling to control the congestion in the network, using application and network information, and global destination buffer status, respectively. HAT adjusts the global throttle rate based on the link utilization at routers if the application intensity, quantified by the L1 MPKI, exceeds a threshold. HAT uses empirically determined parameters for the throttle adjustment epoch length and threshold, which deters the use of those values in our simulation environment as we evaluate different application workloads. We observe that adjusting a global throttling rate does not reduce the starvation at all nodes in a mesh network. Namely the center nodes continue to be starved as flits from other nodes generally route through the center of the mesh under uniform random traffic. The distributed self-learning throttling is more effective as each node's throttle rate varies based on experience and network starvation feedback, and adapts to the workload. The self-tuned congestion control in [77], uses global congestion information to throttle the source, where the threshold is self-tuned to increase throughput without dramatic changes in the latency. However, the global information gathering is costly as congestion and throughput information is sent on reserved side-band channels. Although, we use global information to throttle the source, the feedback network and storage overhead is minimal. RAFT [64] targets power/performance optimization by dynamically tuning the router frequency in response to network load, but this only considers dynamic power consumption and not leakage, which is becoming more prominent.

While some works target specific challenges of bufferless NoCs, e.g. critical path, re-assembly buffers, we specifically target the performance limitations of bufferless NoCs and show that with the appropriate flow control, routing, throttling, bypassing mechanisms, and single-cycle multi-hop links, bufferless NoC performance can indeed reach that of buffered NoCs. The prior proposals are orthogonal to SCEPTER and can possibly be used in conjunction. High throughput performance has not been extracted previously, especially without the use of small side buffers [31]

Buffer Area/Power Optimization. In addition to the work targeted for bufferless networks, area and power reduction of buffers is addressed by countless works. ViCHaR [67] dynamically allocates a central buffer and can realize the same performance with fewer buffers, but requires complex arbiters and control logic, and still requires buffers, whereas SCEPTER is completely bufferless. Elastic Buffer Flow Control [62] uses the buffer space in pipelined channels to reduce the cost of buffers. Kilo-NoC [38] ensures QoS with a low-cost elastic buffering and physical express links achieving similar latency benefits as the SMART baseline.

Hot Potato Routing. Hot potato routing [74] emerged in traditional networks and was found to be useful for optical networks, where buffers are more expensive than deflections [22]. The Chaos router [49], uses deflections as a means for adaptive routing. It eliminates priority routing and uses deflections instead, and also guarantees livelock and deadlock freedom. In [37], an analysis of the network behavior when performing deflection routing in hypercube network reveals that it performs well even under high loads.

Drop-Based. Since we focused on deflection routing, we thus far only considered the prior work related to it. Some bufferless routers advocate the dropping of packets in the event of contention [35]. The SCARAB [40] architecture utilizes a fixed-delay circuit-switched negative acknowledgement network to indicate a packet has been dropped and needs to be retransmitted. SCARAB achieves a higher clock frequency and lower latency than a deflection-based NoC for low loads, but poorer latency and throughput than buffered NoCs.

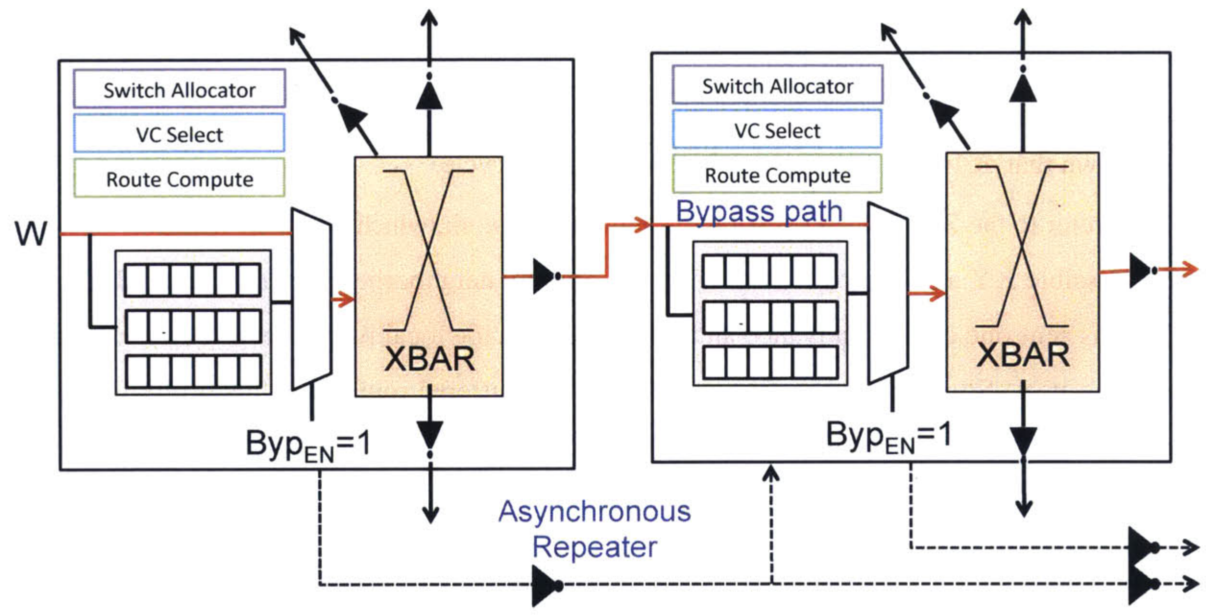


Figure 5-1: SMART NoC Router and Bypassing

5.2.2 SMART Interconnect

Deflection routing, although simple and effective for bufferless NoCs, suffers from performance loss due to excess deflections and the associated latency penalty. Single-cycle Multi-hop Asynchronous Repeated Traversal (SMART) [50] can potentially reduce the latency penalty of deflections to zero even for non-minimal paths.

SMART eliminates the latency dependence on hop count by using clock-less/asynchronous repeaters in routers' crossbars and associated flow control to bypass multiple routers in the same cycle. Figure 5-1 shows how multiple routers can be traversed using preset bypassing signals. A setup request is sent a cycle in advance to arbitrate for the crossbar switch access. In Figure 5-1, the setup request succeeded and successfully asserted Byp_{EN} and allocated the crossbar switch to connect *West* input to *East* output.

Setup Request. Each flit sends a smart-hop setup request (SSR) to the routers along the path to its destination. SSRs are dedicated repeated wires that are $log_2(1 + HPC_{max})$ bits wide, where hops-per-cycle (HPC_{max}) is the maximum distance the flit can traverse in a single cycle. Each SSR carries the number of routers to bypass on the dedicated repeated wire of length HPC_{max}. HPC_{max} is determined by the interconnect and repeater characteristics of the underlying process. In [19], a sensitivity study of the HPC_{max} with respect to clock frequency is performed for SMART's clockless repeated links, carried through post-layout with Cadence Encounter, on IBM 45nm SOI technology. It is shown that at 1GHz, 13-hops can be traversed in a cycle.

We refer to the SMART_2D version throughout this work, which has dedicated SSR links along each possible XY and YX path from each router to all neighboring routers within HPC_{max} hops. A SSR is sent one cycle in advance and arbitrates with the local flits at each router for access to an output port. Similar to a baseline virtual channel buffered router, the flits arbitrate locally to determine the allocation of a flit to an output port. However, in the SMART approach there are additional switch allocation requests that originate from farther nodes. Thus, remote SSRs arbitrate with the local buffered flits for allocation of an output port. Control signals are then setup for the next cycle based on the SSR arbitration outcome. If the SSR wins allocation, the bypass path is enabled such that the flit will route straight through the crossbar switch and move to the next router. If the SSR loses, the buffering path is enabled and the flit will be buffered the next cycle. The flit does not have to wait for a grant signal to be returned indicating bypass success. When a flit arrives at the next router the appropriate control signals are set to reflect whether the setup request succeeded or not.

Switch Allocation and Prioritization. The prioritization mechanism is used to decide which flits are buffered and which win switch allocation. Every router prioritizes SSR requests using

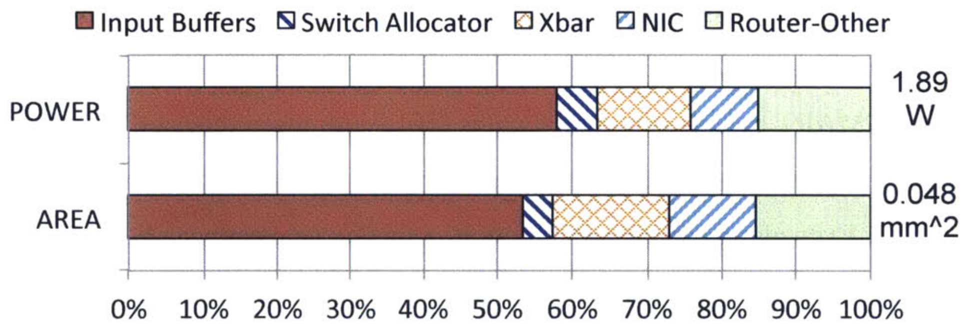


Figure 5-2: SMART NoC Area and Power Breakdown

a fixed priority scheme that is based on the flit's distance from the source. Thus, a *Prio=Local* scheme gives highest priority to the local flit over neighboring flits and those farther away. The single-cycle paths are not guaranteed as the flit may not arrive even if the switch has been allocated. If *Prio=Bypass* is enforced, flits farther away from the source have a higher priority. The same priority is enforced between SSRs to ensure that flits do not traverse farther than HPC_{max} hops in a cycle.

Area and Power Impact of Buffers. The power breakdown of the SMART NoC work in [50] reveals that the buffers consume on average 40% of the total network dynamic power on the IBM 45nm SOI process. The rest of the power is consumed in the crossbar, switch allocation and link traversal, with SSRs taking up negligible power overhead. Figure 5-2 shows the post-layout router area and power breakdown of a 64-node SMART NoC chip with 8 virtual channels (VCs) and 1 flit buffer per VC recently fabricated on a 32nm SOI process. The input port buffers consume more than 50% of the router area and power. The input buffers' absolute power is around 1 W, of which 70% is leakage power alone. Although eight buffers per input port is relatively few compared to commercial NoC prototypes, e.g. Intel TeraFLOPS [43] containing 32 buffers per input port, the power and area overhead is still quite substantial. Bufferless NoCs completely remove such overheads.

5.3 Overview

SCEPTER (Single-Cycle Express Paths and self-Throttling for Efficient Routing) is a bufferless NoC architecture that pushes towards high-performance bufferless NoCs. We lower the average network latency of bufferless NoCs by leveraging single-cycle multi-hop traversals across the network. Multi-hop traversals, routing through multiple routers in the same cycle, are established with the use of SMART clockless repeated paths [50]. With these repeated paths, we are able to set up a route path, and in the next cycle traverse through the preset path, covering multiple hops within one cycle. Even along a deflected direction, a single-cycle path can be potentially traversed that brings the flit closer to its destination.

In a bufferless NoC, flits are constantly in motion and occupying precious links till they are ejected at their destinations. SCEPTER's leveraging of virtual express paths means that flits bypassing from faraway nodes are now thrown into the mix, contending with flits arriving from neighboring nodes, as well as flits waiting in the network interface to be injected. SCEPTER intelligently prioritizes and routes flits from these three sources, opportunistically bypassing on virtual express paths. Going on express paths lower latency, while deflecting flits to idle links get them to their destinations quicker, extending throughput. However, in a bufferless NoC, deflecting flits from one area of a NoC to another shifts congestion to that area, and can deter flits from entering the network (starvation), hurting performance. SCEPTER uses starvation as a key indicator for enforcing livelock freedom and for source throttling. We devise a distributed reinforcement-learning throttling mechanism to control the injection of new flits into the network. Each node independently learns and adapts its throttle rate based on global starvation indicators.

Figure 5-3 shows a SCEPTER deflection-based router with a two-stage switch allocation followed by the switch and link traversal stage. As soon as flits arrive at a router, they are latched into pipeline registers. The next cycle, route computation and switch allocation (*SA-I*) are performed. Latched flits arbitrate for output ports in the first switch allocation stage, where the flits are sorted in order of priority. The highest priority flit has first pick of the output port. A higher priority flit may take a lower priority flit's preferred output port, prompting the lower priority flit to be assigned to another output port. The local flit from the NIC is assigned the lowest priority. At the end of the *SA-I* stage, each latched flit is assigned an output port. If the assigned output port is in a progressive direction, SSRs are sent along the path to destination, up to HPC_{max} hops away, and within a one cycle propagation delay. All routers, in the following *SA-II* stage, arbitrate simultaneously between flits from neighboring routers, SSRs from faraway nodes, and possibly a local injecting flit from the NIC.

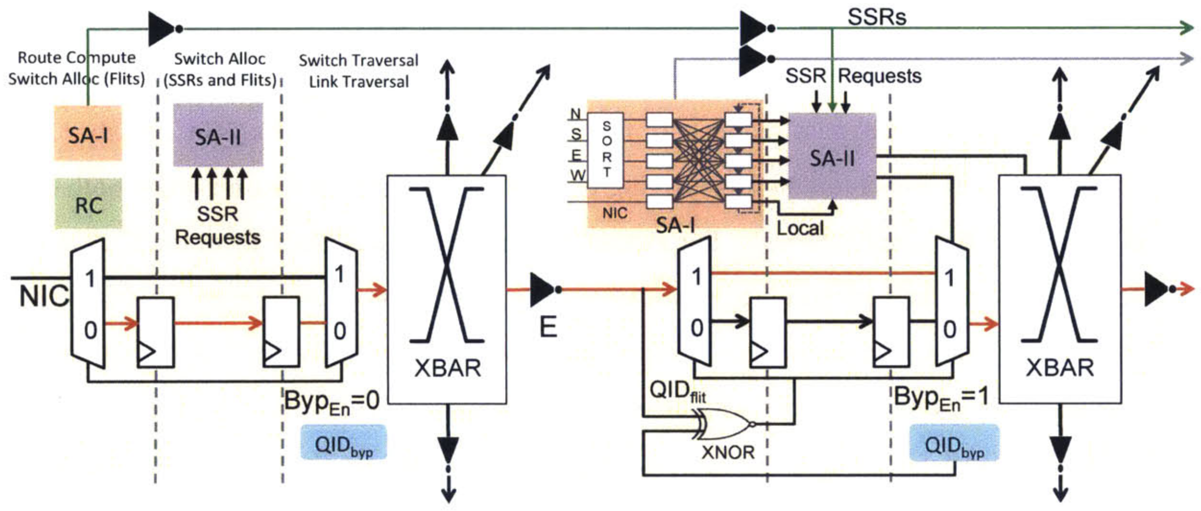


Figure 5-3: Router Pipeline with Bypassing when QID_{flit}=QID_{byp}

In order to leverage single-cycle multi-hop traversals, the SCEPTER router pipeline is similar to SMART router pipeline. Unlike the baseline bufferless router, SCEPTER's switch allocation is split over two stages to arbitrate between local flits and SSRs. SSRs are generated in SCEPTER's first pipeline stage, and take up to a cycle to reach the routers on its path. Since the baseline BLESS router has a 2-stage pipeline and SCEPTER's is three-stages, the multi-hop traversal setup is crucial for obtaining improved performance over the baseline network.

However, for a bufferless NoC, simply adopting SMART flow control will not work. For instance, SMART's *Prio=Local* scheme prioritizes new injecting flits over incoming flits on the N, S, E, W ports and from faraway on the SSRs. With no buffers, and one fewer output port than input ports since the injecting flit will go out of one of the NSEW ports, and flits cannot be deflected through the ejection port, this scheme breaks in bufferless NoCs. As for SMART's *Prio=Bypass*, again, the number of output ports only match the incoming NSEW ports, so injecting flits will not be able to inject except when a flit is being ejected at this router.

5.4 Destination-Proximity Flit Prioritization.

As different flits and bypass requests arrive at the router, the prioritization among them is significant for performance reasons, as is a design challenge as we are pushing towards high-performing bufferless NoCs. A qualitative discussion of few constraints and reasons for the chosen prioritization mechanism is discussed, and fully evaluated with a design sweep in Section 5.9.1.

Input Port Conflict. At each router there is a possibility of receiving up to four flits from the neighboring routers and bypass requests from distant nodes through SSRs. The crossbar switch places an input port constraint on flits arriving from the same input direction as they both cannot traverse the crossbar simultaneously. For example, a flit arrives at a router from the *East* port and is latched in the router's pipeline register. After stage one of switch allocation, it is known that this flit will, say, route through to the North output port. However, multiple SSRs simultaneously arrive from the *East* direction too. The winner among these SSRs will request the *South* output port. Although there isn't a conflict with respect to the desired output port, there is a crossbar input port conflict. The bufferless NoC deflection routing mechanism requires that a flit leaves the router, even if it is along a deflected path. We prioritize flits latched at the router over bypassing flits, hence all SSRs from the *East* direction will fail to setup the crossbar switch at this router.

Prioritization Between Latched Flits. In a bufferless NoCs, the key lies in quickly draining flits at their destinations. Hence, we use a destination proximity (*Dest-Prox*) flit prioritization. Flits are thus prioritized based on the hop count distance from the destination, which is already calculated for route computation and can be obtained with no additional logic overhead. Hence, flits that are closer to the destination are prioritized over flits that are farther. The *Dest-Prox* priority ensures that flits are not excessively deflected once they are in vicinity of the destination node. To arbitrate between flits that are the same priority, i.e. same number of hops from their destinations, a comparison of deflection count is performed and the flit with the higher count is prioritized higher. In the event there is still a tie, the higher priority flit is chosen arbitrarily. Each flit only uses three additional bits for maintaining the deflection count. Every time a flit is deflected, it will increment this count variable. It performs this computation in parallel with route compute at the next router. If the number of deflections exceed 7, for a 3-bit count, the count variable is no longer incremented and remains as is. Only under high congestion and network traffic does the deflection count surpass this range and arbitrary tie resolution ensues. Unlike BLESS' age-based arbitration, *Dest-Prox* avoids the overheads of propagating and sorting timestamps.

Prioritization Between Flit Sources. SCEPTER routers receive flits from different sources: neighboring routers, distant nodes, and local NIC, all generating requests for an output port. If we use an age-based mechanism to arbitrate between the output port requests, the timestamp overhead is not only incurred in the latched and local flits, but the SSRs must propagate this information as well. Each SSR would require an additional $log_2(max\_timestamp)$ bits, where the bound on the timestamp could span into thousands of cycles. This would incur excess wiring overhead for each SSR repeated path.

Flits at the router are given first choice for the output ports due to the input port conflict discussed and network deadlock avoidance. Ensuring that flits are constantly in motion and do not reside in the router pipeline unallocated alleviates deadlock concerns. Any available output ports thereafter are made available to requesting SSRs. If another output port is available but the SSR is not requesting it, the SSR would fail. Latched flits are always assigned an output port, and SSR allocation is not critical for correctness but necessary for high performance. Hence, SCEPTER utilizes the prioritization where *Latched Flits* > *SSRs* > *NIC* at each router.

With flits and SSRs arriving at a router, the local node may not inject a flit into the network if there isn't an available output port for the flit to route to. With the current prioritization scheme, the SSRs do not always succeed because the latched flits have higher priority. Thus, if the SSRs do not schedule the remaining available output ports, the local node has a chance to enter the network. Since SSRs are prioritized over locally injected flits, the network latency is reduced and additional flits do not enter and abruptly congest the network. However, there are starvation concerns when another node's traffic can stall this local node's injection for a long time period. The problem is not just limited to the neighboring routers sending flits to this router, but all the routers within HPC_{max} of this router can prompt starvation. While buffer occupancy is an indicator of congestion in buffered NoCs, *starvation* is the key indicator of contention in bufferless NoCs, and SCEPTER heavily leverages that for output port selection (Section 5.5) and starvation avoidance (Section 5.8), extending throughput.

5.5 Starvation-based Output Port Selection

We identify the key aspects of flit prioritization that exist in the SCEPTER router and describe the *Dest-Prox* arbitration policy between flits, where the flit closest to the destination has the highest priority. SSRs are prioritized over injecting flits from the local node, but are lower priority than the flits at the pipeline registers of this router. Now that the requests are put into a priority order, *how* *do we determine the desired output port for a flit?*

In buffered NoCs, congestion is estimated using buffering information. However, for bufferless NoCs we need another metric. In SCEPTER, we use starvation of the injection port as an indicator of congestion. Specifically, flits are queued at the output of the NIC and dequeued when a local flit is able to inject. If the injection queue depth in the NIC has exceeded a threshold, *IQT* number of entries, it informs the neighboring routers by asserting a one-bit starvation flag. Stopping all neighboring flits from routing to this router will lead to a spreading of congestion to other nodes. We instead propagate the starvation indicator bit to only one of the four input directions and rotate among the ports every cycle. Moreover, the starvation indicator bit changes the prioritization at this router to allow local injecting flits priority over SSR requests when the injection queue depth exceeds *IQT*, easing starvation. For high injection rates/queue depths, the SCEPTER network effectively filters away the starvation effects of remote SSRs, reducing starvation effects to that of neighboring routers, similar to BLESS.

We perform minimal adaptive routing at each router using starvation indicators from neighbor nodes to select a progressive direction. An output port with the starvation bit asserted will not be routed to if another output port is available. Thus, if the progressive directions are congested, the flit may deliberately request a non-minimal output port. In such a case, we observe a ping pong effect, where the request that is deflected in a non-minimal direction will choose to route to the current router again and the flit will move back and forth between the two routers. We toggle the routing policy from XY to YX or vice versa, based on the deflected direction. For example, if a flit with YX routing policy wants to route 2 hops in the positive X direction (East) and 4 hops in the positive Y direction (North) from the current router, and is deflected along the South output port. Since the Y direction is congested, the routing policy is toggled to XY. This means that the X direction is prioritized over the Y direction when performing output port selection.

If a flit chooses a non-minimal output port, more than once, we upgrade the flit to a high priority status and route it to the nearest edge. Along the path to the edge, if a minimal output direction is available, the flit will change to XY or YX routing, and retain its high priority status such that it reaches the destination with a low chance of deflection. This can be viewed as a form of lazy misrouting, in which flits are routed non-minimally only if it fails consistently [78].

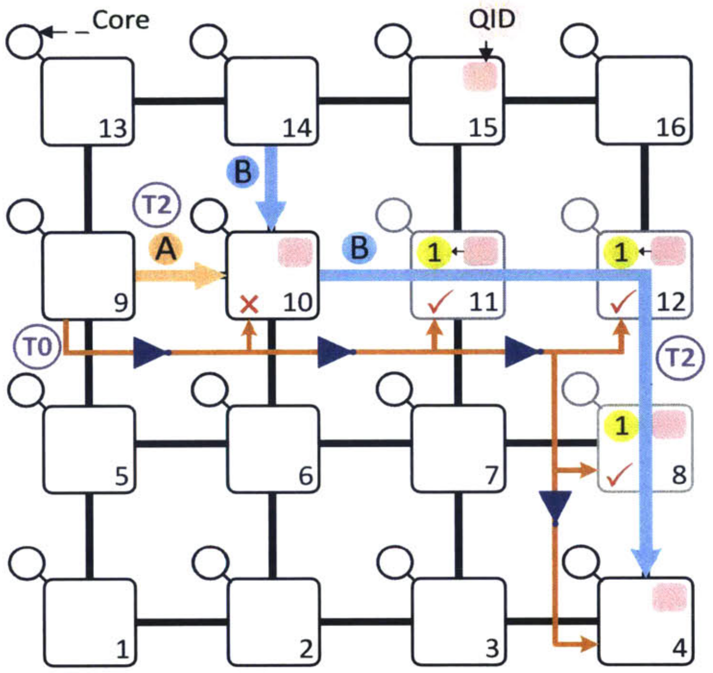


Figure 5-4: Opportunistic Bypassing Example

5.6 Opportunistic Bypassing

The key to network latency reduction is the bypassing mechanism. Many times the SSR succeeds in setting up the crossbar switch at certain routers along the path but fails at other routers as well. Thus, a flit cannot always traverse through the path it dynamically set and the switch setup can be considered wasted. We take advantage of this situation by opportunistically taking scheduled bypass paths that end up being wasted. When a SSR succeeds in allocating the crossbar, a two-bit quadrant ID, $\mathrm{QID}_{\mathrm{byp}}$, is recorded pertaining to the destination quadrant of the SSR. The network is divided into four equal-sized quadrants, where $QID = 00$ indicates the top right quadrant, $QID = 01$ is the top left quadrant, $QID = 10$ is the bottom left quadrant, and $QID = 11$ is the bottom right quadrant. In the following cycle, a flit arrives and is able to bypass the *SA-I* and *SA-II* stages and directly traverse the crossbar. However, another flit may arrive, one that did not explicitly preset the crossbar switch with a SSR. Hence, a check is performed on whether the destination quadrant ID, $\mathrm{QID}_{\mathrm{flit}}$, of the incoming flit matches the $\mathrm{QID}_{\mathrm{byp}}$ value, as shown in Figure 5-3. Figure 5-4 shows an example that demonstrates the key idea behind opportunistic bypassing.

1. At time T0, Flit A sends a SSR from source node 9 along the path to the destination node 4. Flit B arrives at node 10 and is latched.

2. At T1, the SSR arbitrates at routers 10 and 11 to allocate the crossbar from *West* to *East* for the next cycle. At router 12 it requests the crossbar from *West* to *South*, and router 8, from *North* to *South*. The SSR succeeds at all intermediate routers except for router 10, indicated by the cross and ticks in the Figure. A single-cycle path has been preset from router 10 to 4, with $QID_{byp} = 11$.

3. At T2, Flit A is latched at node 10. Flit A's bypass fails as the SSR did not succeed.

4. At T2, Flit B arrives at router 11 and compares its destination quadrant ID, $QID_{flit} = 11$, with the SSR's quadrant ID ($QID_{byp} = 11$). After simply performing a bitwise XNOR, the flit decides to take the bypass because the QID matches. Flit B is able to take the single-cycle express path to router 4.

5. At T3, a SSR request from another flit arrives from the *East* direction at router 10 and is requesting the path from router 10 to router 2. This SSR request arbitrates with Flit A in the *SA-II* stage. Since Flit A and the SSR request both arrive from the same input port direction, the SSR fails due to the *input port conflict*, and Flit A successfully allocates the *East* output port.

Flits are able to hop onto these idle bypass paths opportunistically and effectively reach closer to the destination node. The QID_{byp} is useful for ensuring the multi-hop paths taken do not unnecessarily deflect the flit in non-progressive directions. Once the flit arrives in vicinity of its destination, it will be prioritized higher than some other flits due to the *Dest-Prox* prioritization.

5.7 Rotating Highest Priority Source ID

Latched flits are always prioritized over SSRs and locally injected flits from the NIC. This policy ensures a flit in the router always exits after moving through the router pipeline. Deadlock is avoided as a result, as routing cycles do not form within the SCEPTER bufferless network. However, livelock may occur as certain flits roam constantly, continuously deflect, and are not serviced and sent to their destinations.

SCEPTER request prioritization mechanisms do not guarantee livelock freedom unlike the age-based mechanism of BLESS. In order to enforce livelock guarantees, all routers have to follow consistent ordering rules. This way the flit is prioritized globally above the other flits and SSRs, allowing it to reach the destination within a fixed-latency bound. We maintained synchronized time windows at each node in the network, and enforce a *consistent* highest priority source ID each time window. All flits in the network that originate from this source are prioritized over other flits. The highest priority source ID rotates each time window. Since each node may have multiple outstanding requests in the network, it is possible for two or more highest priority (HP) flits to contend for the same output port. In such cases, the HP flit with the highest deflection count will succeed, while the others will be deflected, with ties arbitrarily resolved. Since each source will become HP in time, livelock freedom is ensured.

Time Window. The length of the time window should be sufficient to drain the network of a request from this source. Since these HP flits are allowed to schedule multi-hop traversals with highest priority, a HP flit is able to reach the destination in one cycle, provided it is within HPC_{max} hops away. Thus, the minimum time window is 2 cycles, one cycle to reach the destination node and one cycle to obtain the grant for the ejection link to the network interface. In larger networks, it may take more than 2 cycles to reach the destination as multiple multi-hop traversals need to be setup. The time window lower bound is the maximum Manhattan distance divided by HPC_{max}. However, in high congestion cases the SSR may not be able to set up a single cycle path to the destination node unless flits are temporarily stored in the pipeline register while the HP flit bypasses and potential deadlock cases are appropriately addressed. At each hop, the HP flit is prioritized higher than other flits and is always granted the desired output port and is not deflected. The upper bound on the time window is the maximum Manhattan distance times the per-hop latency: 16 cycles for a 64 node network, 32 cycles for a 256 node network.

SSRs. When a SSR is received, the highest priority source ID is checked for this time window. If the source ID matches, the router prioritizes this flit over the latched flits, but only if there is a free output port that the latched flits can deflect to. If not, the SSR fails as we always ensure latched flits are allocated an output port. If a flit arrives with a source that matches the highest priority source

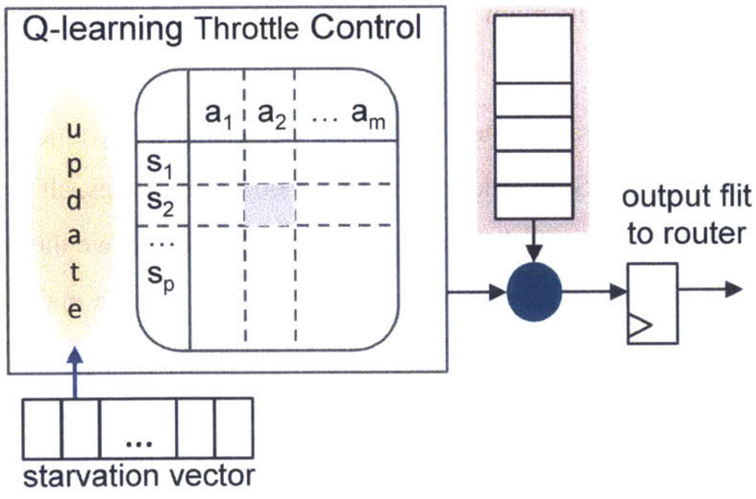


Figure 5-5: Self-Throttling with Q-Learning

ID, its priority supercedes the *Dest-Prio* order.

5.8 Self-Learning Throttling

In a general bufferless or buffered mesh network, only the neighbors are responsible for creating congestion at a particular router. However, in the SCEPTER NoC, all nodes within HPC_{max} hops away are potentially responsible for congestion and thereby prevent local flit injection. As flits are constantly moving in the network, congestion in a bufferless NoC shows itself through starvation – when nodes cannot inject. We extend the starvation indicators used in each SCEPTER router for output port selection and highest priority source selection, sending these indicators globally to all nodes. We then devise a self-learning throttling mechanism that adapts based on this global network starvation status, in order to improve bandwidth allocation fairness, reduce network starvation rate, and extend network throughput.

Global Starvation Status. We propagate global starvation information using a separate bufferless starvation network, adapted from the notification network in [28]. All network interfaces monitor the injection queue depths, and if the depth exceeds *IQT* entries, a starvation flag is set. The starvation indicator is sent on the starvation network as a one-hot N-bit vector, where N is the number of nodes in the network. The bit in the vector corresponding to this node's ID is asserted if the starvation flag is set. In an 8×8 network, the starvation network will take the maximum

Manhattan distance of 16 hops to ensure the global information is propagated to all nodes. Using synchronized time windows of length *L*, each node will propagate the starvation information at the beginning of a time window. For an 8×8 network, the time window length, *L* can be set at 16 cycles as one hop in this congestion feedback network incurs a latency of one cycle.

Throttling. Source throttling is a common technique used for congestion control in networks [18, 77]. Every *L* cycles, the feedback network provides a view of the global starvation state of the network. Using this information, we determine when and how to throttle the sources. The most straightforward approach is to stop non-starved nodes from injecting and allow the starved nodes to continue attempting to inject into the network. We call this approach, *ON/OFF Throttling*, however it leads to increased starvation in the stopped nodes. To be able to find the optimal action based on network feedback, we use reinforcement-learning to adapt and learn from the starvation state and prior experience on throttling rate adjustments.

Q-learning. Q-learning is a reinforcement learning technique where the agent/nodes try to learn the optimal action selection policy. Essentially this means that based on the environment interactions, the Q-learning algorithm will, over time, determine the best action to take, given an environment state. Each node maintains a table of Q[S,A], where S is the set of states and A is the set of actions.

We maintain a Q-table of 8 states and 5 actions at each network interface. The states correspond to the change in starvation across the network and also account for any change in the local node's starvation status. The five actions correspond to increasing, decreasing, or retaining the throttle rate. If the throttle rate is to be increased, there are two possible actions, either increase slowly with a smaller increment value (in our case it is 1), or increase quickly with a larger increment (in our case it is 2). Thus, an action that increases the throttle rate by one, would stop one flit from injecting into the NoC in a ten-cycle time interval.

The Q-table, as seen in Figure 5-5, is initialized to all zeroes. The state s of the network is observed and a action a is selected to be carried out. After the action is performed, the current state $s^{'}$ is observed. The Q-table value is updated according to Equation 5.1, where α is the learning rate, γ is the discount rate, r is the observed reward, and $max(Q^{'}[s^{'}, a^{'}])$ is the maximum value in the Q-table for the current state, $s^{'}$. In Figure 5-5, the shaded entry corresponds to the Q value with the current state, s_2 and action a_2, and pertains to the maximum Q value for the network state. The

Table 5.1: SCEPTER Network Parameters

NoC Topology	6×6, 8×8, 16×16 mesh
Routing	Minimal Adaptive, Non-minimal priority
Flit Prioritization	Destination Proximity
Request Prioritization	Flit > SSR > NIC
HPC_{max}	8
Time Window (L)	12 (6×6), 16 (8×8), 32 (16×16)
Injection Queue Threshold (IQT)	20 (8×8), 40 (16×16)
Learning Rate (α)	0.1
Discount Rate (γ)	0.1

global congestion vector from the feedback network triggers an update of the Q-table entry, where the learning rate indicates how quickly new congestion information is factored into the update. The future reward potential is computed by obtaining the maximum Q value for the current state and factoring it into the prior state's Q value update, where the discount rate indicates the importance of the future reward.

$$Q[s,a] = (1-\alpha) * Q[s,a] + \alpha * (r + \gamma * max(Q'[s',a'])) \tag{5.1}$$

The reward value is assigned based on how the starvation count changes from the previous state. If starvation increases, a negative reward is given, and if the starvation drops, a positive reward is given. After some interaction with the environment, the Q-table will be populated with values. Using the Q-table values, an action can be selected that will either increase, decrease, or retain the throttle rate at the node. For an observed starvation state, the Q table is checked to find the action that has the largest Q value, meaning it has potential to give the maximum reward. This action is taken and the value-iteration of Q is continually performed. To allow for continual learning, the controller switches to a random action selection every one in ten iterations, such that other actions and network states are explored.

5.9 Architecture Analysis

We evaluate SCEPTER and baseline architectures with the gem5 [15] cycle-accurate simulator and the GARNET [10] network model. The evaluated system parameter settings are shown in Table 5.1.

The baselines for performance comparison are the BLESS NoC and SMART virtual-channel buffered NoC, described in depth in Section 5.2. The BLESS NoC serves as a lower bound on the performance of SCEPTER as it is the best-performing bufferless NoC. The SMART state-of-the-art buffered NoC obtains high-throughput and low-latency, and forms the upper performance target.

BLESS uses an age-based prioritization mechanism to ensure oldest flits are continually drained from the network. As for the SMART NoC, packets follow deterministic XY routing and use SSRs to setup multi-hop traversals. HPC_{max} is set to be the same (8) for SMART and SCEPTER for proper comparison. SCEPTER's performance is evaluated for both synthetic and full system applications. First, a design sweep of the different architecture choices is discussed. Next, a network-level and full system applications performance comparison is discussed with respect to the baselines.

5.9.1 Design Space Exploration

The architecture settings for the SCEPTER NoC are chosen by performing a design sweep of the request prioritization, flit prioritization, opportunistic bypassing and adaptive routing using synthetic traffic patterns. For SCEPTER's request prioritization exploration, we are unable to evaluate the exact *Prio=Local* or *Prio=Bypass* mechanism from the SMART NoC as it requires buffers to store unsuccessfully allocated flits. However, we evaluate feasible alternative mechanisms for the request and flit prioritization. Network-level synthetic traffic simulations allow us to observe the saturation throughput of SCEPTER in comparison to the BLESS NoC and the high-performing SMART NoC with varying buffer counts.

Impact of Flit Source Prioritization

At each router, there are two levels of prioritization. The first level is between the flits arriving from the neighbors, and second is between all the switch request sources: SSRs, latched flits, and injecting flit at the NIC. We discussed that prioritizing the SSRs over all other requests would lead to a potential starvation as latched flits will have to reside in the pipeline registers and may be continuously overtaken. Hence, the latched flits always have the highest priority, but we are able to characterize the effects of prioritizing either the SSRs or NIC's local flit, first. We obtain the average

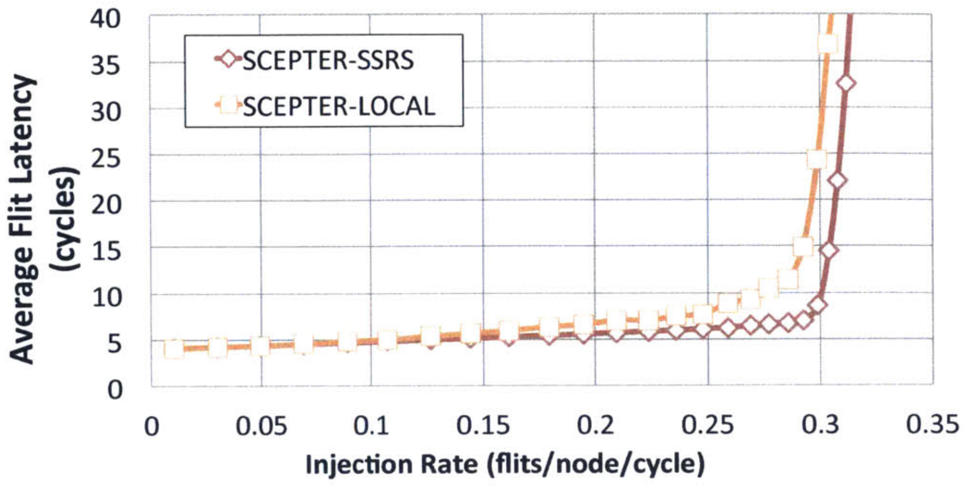


Figure 5-6: Flit Source Prioritization (64 Nodes) for the Tornado Synthetic Traffic Pattern

latency as the injection rate is varied for both prioritization approaches, while keeping all other settings the same. Figure 5-6 shows the results when we compare the average latency of SCEPTER-SSRs (SSRs prioritized over the local flit) to SCEPTER-LOCAL(local flit prioritized over SSRs). The SCEPTER-LOCAL option saturates more quickly as the network is being populated with more flits, and the SSRs begin to fail due to congestion.

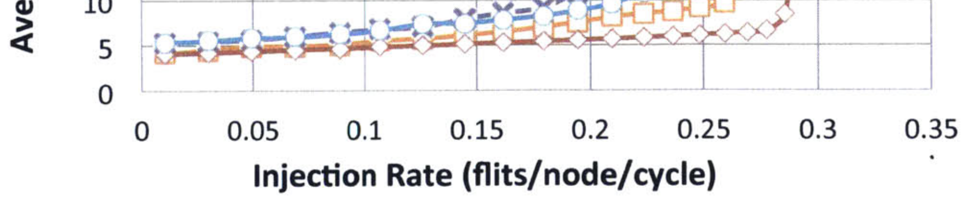


Figure 5-7: Latched Flits Prioritization (64 Nodes) for the Tornado Synthetic Traffic Pattern

Impact of Latched Flits Prioritization

With the request prioritization approach now fixed at prioritizing SSRs over local flits, we determine the best prioritization among flits that arrive from neighboring routers. BLESS uses age-based arbitration, but we observe that age-based arbitration requires an expensive timestamp to be sent per flit. Moreover, if all the requests arbitrate according to age, it requires SSRs containing timestamps to be sent across the entire network, which is highly impractical.

Figure 5-7 shows the average latency as the injection rate is increased for uniform random traffic. The Defl prioritization considers a count within the flit that contains the number of deflections the flits has encountered. The higher the deflection count, the higher the priority. The Age prioritization uses the flits' enqueue time to determine the age. The older flit has a lower enqueue time, and hence a higher priority. The other two are distance-based approaches. Src-Dist uses the flit's distance from the source, in terms of hops, such that flits farther from the source are allowed first pick on the output port. Dest-Prox performs the opposite prioritization, where the flits closer to the destination are prioritized above others. For two or more flits with the same priority, one is chosen arbitrarily for the Defl and Age mechanism; although for distance-based approaches the deflection count is the tie breaker.

The results clearly show that the destination proximity mechanism outperforms the rest. It addresses a frequent case that leads to higher average network latencies: if a flit is moving through the network and is close to its destination, it may be deflected in any of the other prioritization approaches, only to try again and again until the livelock mechanism prioritizes this flit's source ID, occupying precious link bandwidth.

Adaptive Routing and Opportunistic Bypassing

The benefits of two features, adaptive routing and opportunistic bypassing, to the throughput gains of the SCEPTER NoC can be seen in Figure 5-8. Each feature contributes only a minute improvement in saturation throughput. Opportunistic bypassing isn't effective alone because the flits will keep attempting to route through minimal paths, which tend to be highly congested at the center of the mesh. SSRs would tend to fail along these routes, primarily because of congestion. Since other paths are not explored, the number of single cycle express paths that are successfully set up

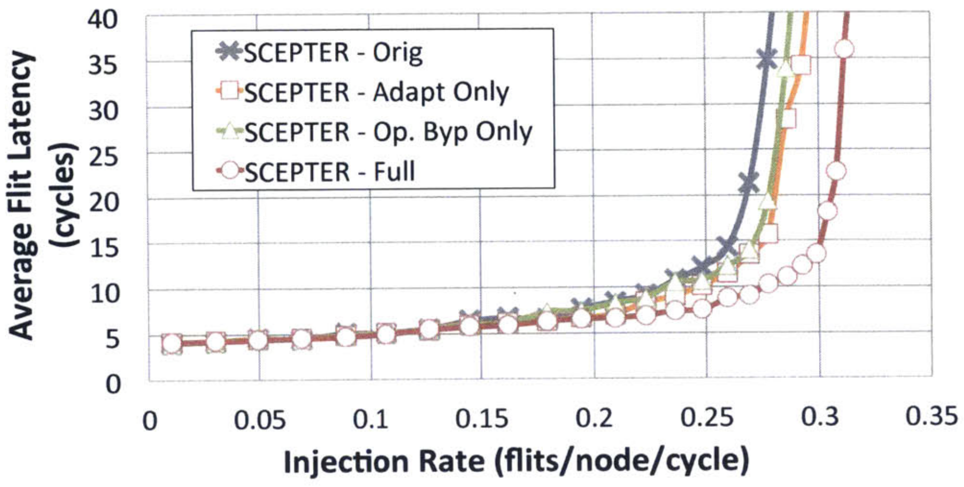


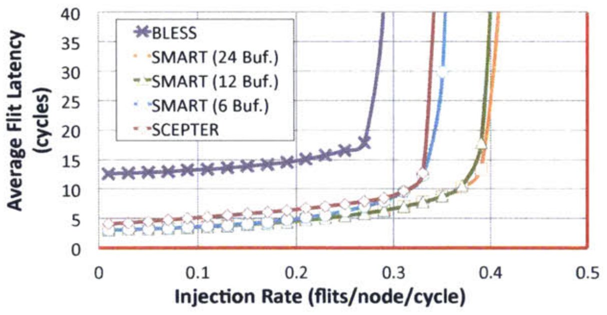
Figure 5-8: Bypassing & Adaptive Routing (64 Nodes) for the Tornado Synthetic Traffic Pattern

drops sharply with injection rate. Similarly for adaptive routing, a flit can route along minimal and non-minimal paths, but can only hop across multiple routers if it explicitly requested the path and successfully allocated it. Thus, a combination of both features is the most effective at improving performance.

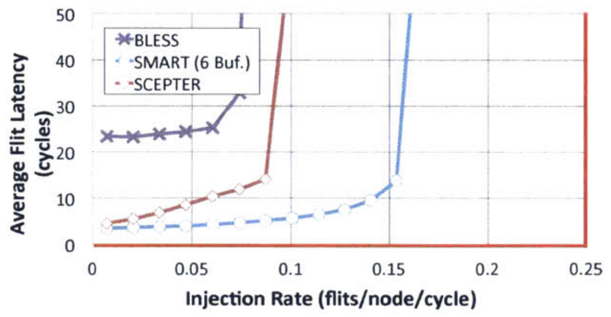
5.9.2 Performance with Synthetic Traffic

In Figure 5-9 the BLESS NoC saturates very quickly for all the synthetic traffic patterns. For uniform random traffic, SCEPTER achieves a 60% reduction in latency, and $1.2\times$ higher throughput as compared to the BLESS NoC. Thus, SCEPTER is able to reach the SMART buffered network, 6 buffers, per input port, i.e. 30 buffers per router, without additional complexity and a complete elimination of buffers in the network. Taking the average across the traffic patterns, SCEPTER reduces the average latency by 62% and achieves a $1.3\times$ higher saturation throughput. This throughput gain is obtained by taking the ratio of the saturation load of SCEPTER and BLESS.

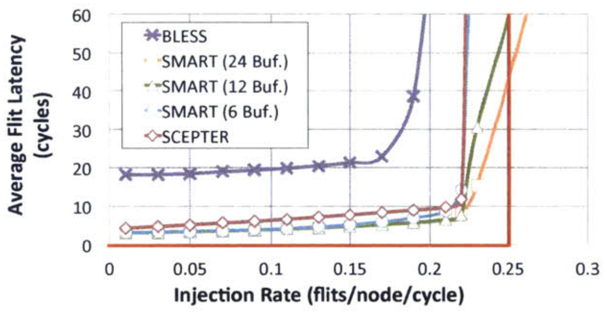
For the 64 node network, under uniform random traffic and at saturation, the percentage of allocated crossbar switches granted to SSRs is only 10%. Of which, 53% are traversed by a flit, and 47% are wasted switch allocations. At this operating point, the express path achieves on average 1.5 hops; a drop from the average 5 hop bypass path for low network load. This behavior is attributed to the prioritization scheme that is necessary for a bufferless NoC. At each router the latched flits are prioritized higher than SSRs and local flits. The drop in SSR switch allocation indicates that the



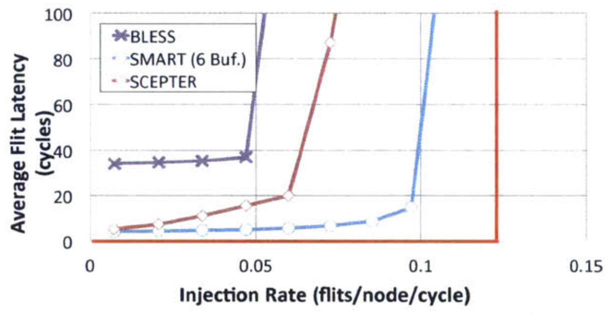
(a) Uniform Random (64 nodes)



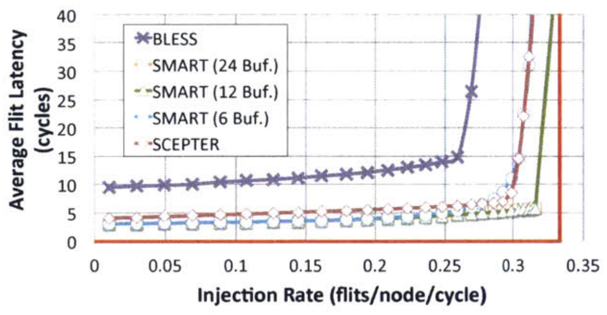
(b) Uniform Random (256 nodes)



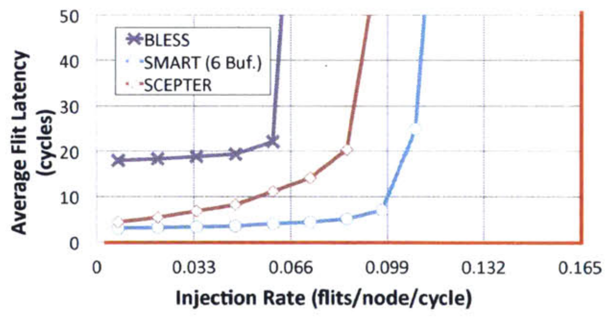
(c) Bit Complement (64 nodes)



(d) Bit Complement (256 nodes)



(e) Tornado (64 nodes)



(f) Tornado(256 nodes)

Figure 5-9: Network-Level Performance for 64 and 256 Nodes

switches are continually being allocated by latched flits instead, thus preventing SSRs and local nodes from accessing the crossbar.

Congestion is the culprit as it causes a drop in network performance and an increase in starvation. The livelock mechanism is meant to relieve the network of some congestion but has a lower release rate than the injection rate, i.e. at least 1 flit is prioritized and ejected every 16 cycles for the 64 node network. However, if the sources keep injecting at a faster rate, the congestion cannot be controlled by the livelock mechanism alone, especially when scaling the network size. Figure 5-9 shows the performance for a 16×16 network. The gap between the buffered and bufferless network is now more apparent, however SCEPTER outperforms the baseline bufferless NoC and achieves a $1.3\times$ higher saturation throughput and 62% reduction in latency.

$$F = \frac{(\sum_{i=0}^{n} x_i)^2}{n * \sum_{i=0}^{n} x_i^2} \tag{5.2}$$

5.9.3 Starvation and Fairness

We evaluate the starvation and fairness characteristics and the impact of the distributed reinforcement-learning throttling mechanism. The hotspot traffic profile, where 75% is uniform random traffic and 25% is traffic towards a center node, allows us to observe the effects of unfair network access due to network congestion. We determine the learning rate, *IQT* and discount rate empirically, and these values, in Table 5.1, are kept constant throughout the simulations.

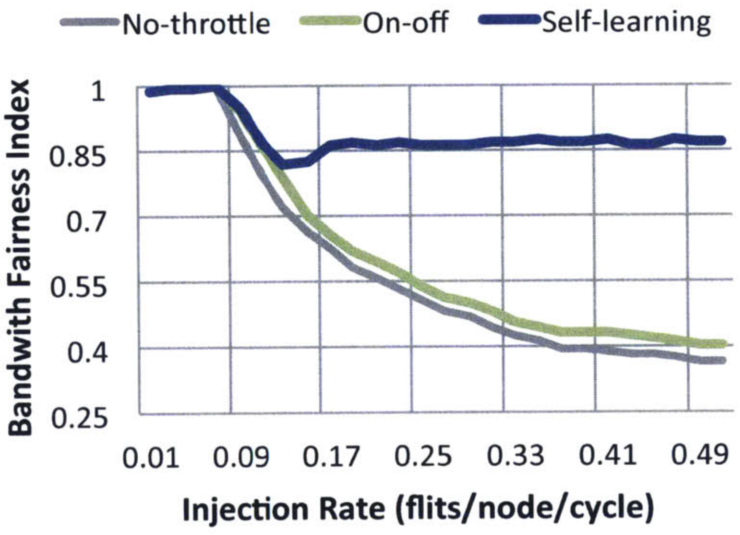
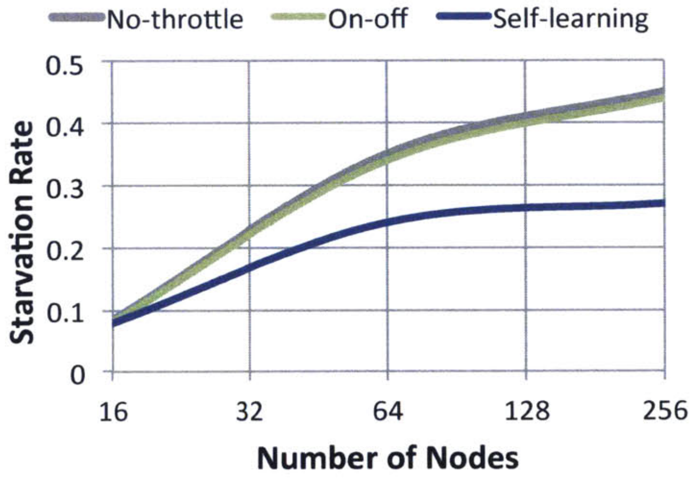


Figure 5-10: Bandwidth Fairness for 64 Nodes as the Injection Rate is Varied

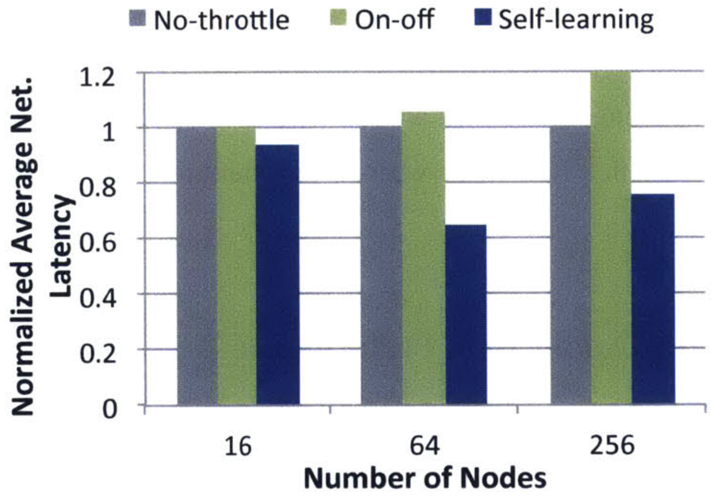
Bandwidth Allocation Fairness

We vary the injection rate of the hotspot traffic profile for 64-nodes and observe the bandwidth allocation fairness. While a lower starvation rate in Figure 5-11a implies fair network access, we obtain the network bandwidth allocation fairness using the per-node throughput to determine the overall fairness of the network. We use Jain's fairness index [39, 45], in Equation 5.2 where x_i is the throughput of the ith node in the network. An index value of 1 indicates that 100% of the network is receiving equal bandwidth allocation.

As seen in Figure 5-10, SCEPTER with *ON/OFF Throttling* enabled does not provide significant fairness improvement. This method causes starvation to shift to other nodes in the network as they may be stopped from injecting for an entire time window. If the node is stopped from injecting



(a) Starvation Rate



(b) Average Network Latency

Figure 5-11: Impact of Core Count on Network Latency and Starvation Rate

flits this time window, the injection of flits can be enabled at the earliest during the following time window. Thus, for at least 16 cycles the flits from the node will accumulate in the node's injection queue and potentially become a starved node. SCEPTER with *Self-Learning Throttling* enabled maintains a fairness index of 87% at high injection rates as well. This results in a 77% higher fairness across the network at high injection rates.

Starvation Rate

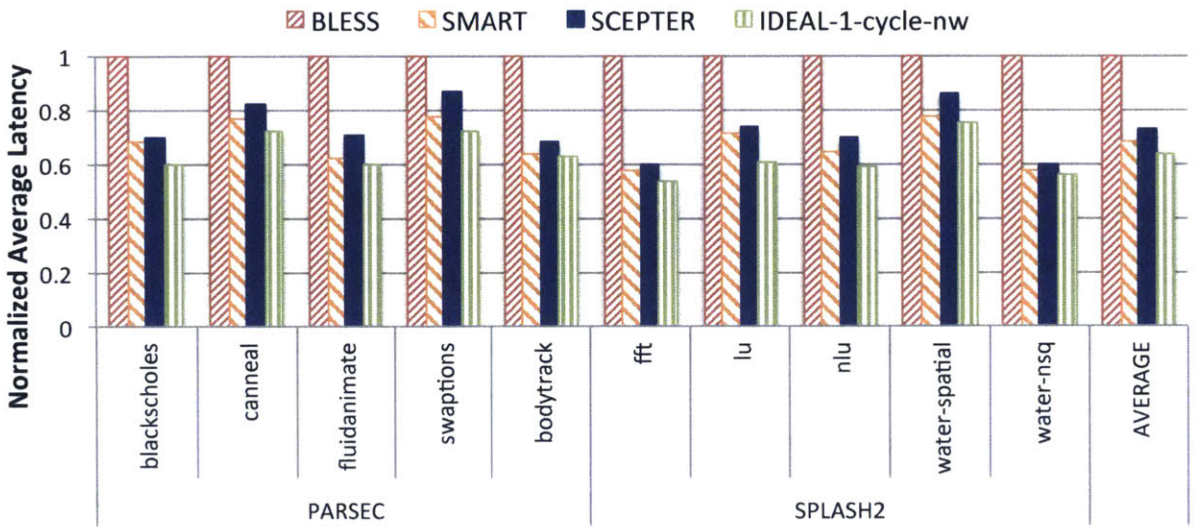
The starvation rate of the network is the average fraction of time when network access is not granted. We specifically evaluate the starvation at each node by taking the maximum number of consecutive cycles, within a time period, when a node is unable to inject a flit into the network. We average it across all nodes in the network, and vary the number of nodes up to 256. Figure 5-11a shows that the self-learning throttling mechanism results in a 31.4% and 38.6% lower starvation rate for 64 and 256-nodes, respectively. The throttling does not affect 16-nodes as a result of a smaller network diameter, HPC_{max} of 8 hops in one cycle, and less congestion as a result.

Average Network Latency

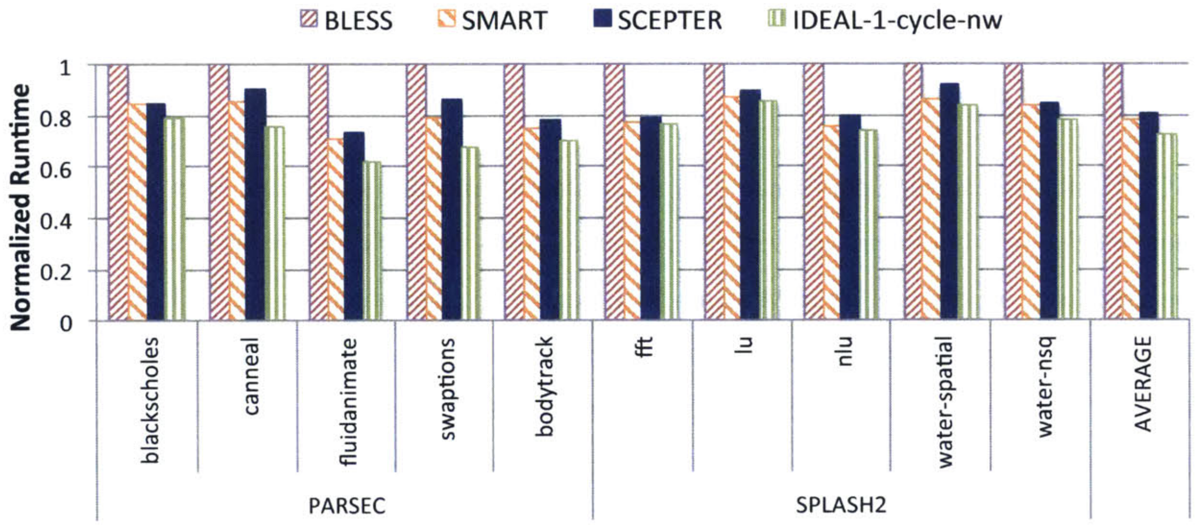
Figure 5-11b shows the reduction in network latency as a result of the self-throttling mechanism, for 16, 64 and 256 nodes. As we vary the number of nodes, we see the effects of the throttling mechanism are more pronounced, with a 34.7% and 24.3% reduction in average latency for 64 and 256 nodes, respectively. For a 16 node network, the network congestion is quickly alleviated with the opportunistic bypassing mechanism and HPC_{max} of 8, which results in close to single-cycle network latencies. Thus, in a 16 node network, source throttling only reduces the network latency by 5.7%.

5.9.4 Full-System Application Performance

For full-system architectural simulations of SCEPTER, we model 36 x86-64 cores in gem5, with split L1 32 KB I/D caches, and private 1 MB L2 caches. SCEPTER and all baselines use the MOESI directory protocol. Figure 5-12a shows the normalized average latency of SCEPTER in comparison to SMART (6 buffers per input port), BLESS, and an ideal one-cycle network. On



(a) Average L2 Miss Latency



(b) Application Runtime

Figure 5-12: Normalized Performance of Full System PARSEC and SPLASH-2 Applications

average, SCEPTER shows 27% lower network latency than BLESS and comparable performance to the SMART buffered network. The average latency incorporates both the network and source queueing latency. For these applications, the L2 miss rate is moderate, however some applications exhibit behavior where larger network latencies are observed, e.g. canneal, where the constraint on the ejection links results in more contention, more deflections, and higher average latencies. SCEPTER is beneficial at reducing the network latency and routing around congested areas, but

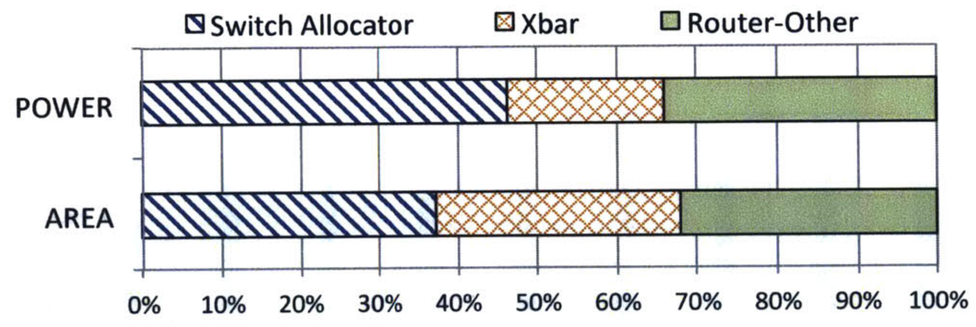


Figure 5-13: SCEPTER NoC Area and Power Breakdown

contention for ejection ports cannot be avoided and results in smaller network latency reduction. Figure 5-12b shows SCEPTER achieves on average a 19% lower application runtime than BLESS, and comparable performance to SMART, with 6 buffers per input port, and an ideal single-cycle contention-free network.

5.9.5 Overheads

SCEPTER achieves high performance by using simple prioritization mechanisms, a livelock freedom mechanism that incurs no overhead, and small SSRs that only require an additional 2-bits for opportunistic bypassing. On IBM 32 nm SOI technology, we synthesize the RTL with Synopsis Design Compiler to obtain the critical path; identified as the multi-hop traversal path. The opportunistic bypassing logic contributes an additional 25 ps per hop to the critical path. Thus, for a HPC_{max} of 8, the clock period is 1.8 ns, 12% higher than the baseline buffered SMART network. Prior work targets this allocation delay [30], but it is not necessary to incorporate such optimizations unless a lower HPC_{max} is targeted and a higher low-load latency is acceptable. The post-synthesis area of the SCEPTER router RTL is 36% lower than the SMART router and 29% lower than a two-cycle buffered router, where both baselines have 8 buffers per input port. Figure 5-13 shows the post-synthesis power and area breakdown of the SCEPTER router. The SCEPTER router consumes 33% less power than the SMART router, although the switch allocation logic is slightly more complex. The processor tile power is about 17% lower with the SCEPTER router instead of the buffered SMART router, assuming each tile contains a simple in-order core, 16 KB split I/D L1 cache, and 128 KB L2 cache.

We utilize a Q-table for the self-learning throttling mechanism at each node to hold the Q values. Since the hot-spot traffic profile results in high congestion, the throttling actions are more fine-grained and allow multiple increment and decrement granularities. The Q-table with 40 entries is shown to be very effective at reducing the network starvation effects for the hotspot traffic profile. Each Q value can be a single-precision floating point value that requires 32-bits to represent. The total overhead of each table is estimated to be 160-bytes, if floating point is used. The global starvation feedback network is adapted from the SCORPIO notification network, where it was shown to be a low overhead network. For other workloads and traffic patterns, the number of Q-table entries can potentially be reduced and should be application and target system dependent. We leave the application-adaptive and network starvation-aware self-learning throttling for future work, where the table size and overheads are determined and optimized with regard to all types of workloads and a range of system configurations.

CHAPTER 6

SC^{2}EPTON: SNOOPY-COHERENT, SINGLE-CYCLE EXPRESS PATH, AND SELF-THROTTLING ORDERED NETWORK

Broadcast and multicast communication plays a role in both message passing and shared memory paradigms. With packet-switched NoCs replacing shared bus interconnects, the broadcast/multicast support is not inherently supported which prompted researchers to add message forking within the network. The multicast forking can take place at the NIC, or within the routers as the multicast propagates through the network. The forking at the NIC creates many unicast flits to replace the single multicast flit, in turn leading to high network congestion even at low injection loads. Solutions to mitigate congestion are addressed in prior research [7,46,51,52,69,80], mostly through the ability to fork flits within the routers. In essence, a single multicast enters the network and at each router, multiple flits are generated and sent out output ports towards the destination nodes. While these proposals target buffered networks to enable the message forking within the routers, none specifically address broadcast/multicast communication in bufferless on-chip networks.

Snoopy coherence places high bandwidth requirements on the interconnect for high-performance gains. To scale the network and keep satisfying bandwidth demands, the VCs or buffers need to scale with core count. The SCORPIO architecture removes the VC allocation from the critical path, eliminating the impact on the router critical path timing. However, area and power still pose a concern, as significant VCs and buffers are needed to handle SCORPIO's large broadcast bandwidth at high core counts. SCEPTER bufferless architecture brings a ray of hope as high performance is extracted through a series of prioritization, routing, bypassing and throttling mechanisms, maximizing opportunities to zoom along virtual express paths while pushing bandwidth. However, SCEPTER is geared for unicast communication and supports broadcast communication by sending multiple unicast messages, each with a single destination. For the full-map directory-based coherence, SCEPTER alone suffices for performance improvement. Snoopy protocols, though, require optimized broadcast communication. With just SCEPTER, each broadcast request spawns N unicast requests, thus even at low to moderate injection loads, the explosion of deflections degrades performance as links are fully occupied. In addition to the performance inefficiencies, the multiple unicasts result in higher network power, thus prompting power efficient and broadcast optimized architectures. Assuming livelock freedom is maintained, via a rotating source ID mechanism or age-based per-hop arbitration, broadcasts eventually reach all nodes. However, the time to deliver the broadcast to all the destination nodes, may be quite large in the SCEPTER network during high congestion, the draining of flits relies on the livelock mechanism. Thus, the flit may continue deflecting until the source ID is prioritized above all others.

6.1 Motivation

Broadcast communication is a a heavy burden for buffered networks and places an even heavier burden on bufferless networks, especially if high performance goals must be attained within a low power envelope. To further illustrate the need for a specific architecture for broadcast communication on bufferless networks, we plot, in Figure 6-1 the average percentage of flits that failed more than once to leave the network (eject to the NIC and core). This is due to multiple flits contending for access to the same output link to the local NIC. We observe that the percentage of flits that encounter ejection contention saturates quicker for broadcasts than unicasts. It saturates at approximately

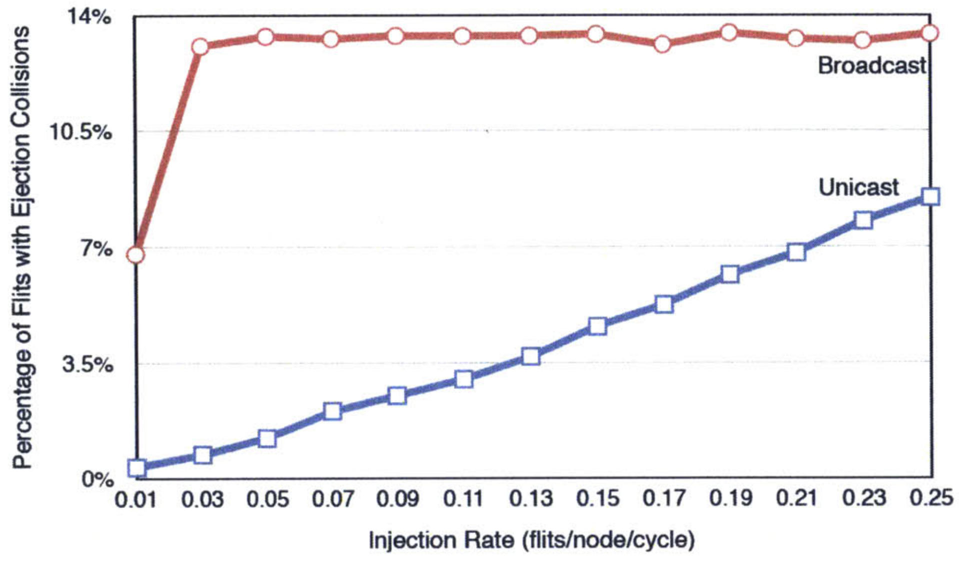


Figure 6-1: Percentage of Flits with Multiple Failed Attempts to Exit the Network

14% as the injection of new flits into the network is throttled by the network injection policy, only allowing new flits to be injected if a free output port is available.

The theoretical limit of unicast and broadcast traffic in a k$\times$k mesh NoC is shown in Table 6.1. Using the Bernoulli process of the rate R, to determine the uniformly distributed, random, destinations for unicasts, and sources for broadcasts. The bounds are determined based on the completed message transaction, from the initiation at the source NIC until the flit is received at the destination(s). Unicast derivations for latency and throughput follow the technique mentioned in [26]. For broadcast traffic, the *broadcast latency*, or time until all destinations receive the flit, is equal to the time for the flit to reach the farthest destination. The channel load is analyzed across the bisection and ejection links [68], where the maximum of the two determines the maximum throughput. For k$\times$k mesh NoC, with k=4, the bisection and ejection load is equivalent. However, for k>4, the ejection links constrain the throughput for broadcast traffic.

The bufferless broadcasting mechanism must (1) ensure timely delivery of broadcast messages to all nodes, (2) minimize the endpoint buffering requirement, and (3) support global ordering for broadcast-based coherence protocols, while keeping ordering latencies minimal.

Table 6.1: Latency and Bandwidth Limits of a k×k Mesh NoC for Unicast and Broadcast Traffic. [68]

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

6.2 Related Work

We briefly review the research on multicast and broadcast communication in buffered on-chip networks. While, bufferless broadcasting and multicasting is not extensively addressed in literature, we discuss a few relevant works, including one that target optical interconnects. In addition to the work in bufferless routing in mesh NoCs, mentioned in Chapter 5, we discuss here, prior proposals which utilize rings for coherent communication.

6.2.1 Multicast and Broadcast Communication in NoCs

Multicast routing is especially challenging as the flit must be delivered to multiple destination nodes. Three general approaches are unicast-based [61], tree-based [7,46,80], and path-based routing [27]. The unicast-based scheme separates the multicast message into M different unicast messages, where M is the number of destinations in the multicast. This approach creates congestion at the source and destination links, and floods the network resulting in high contention and increased packet latency. Tree-based routing creates a tree, with the source node as the root, and destinations on the branches. Virtual Circuit Tree Multicasting (VCTM) [46] sends a packet to build the tree prior to sending the multicast flit. However, VCTM requires additional storage per node to maintain the tree information, and the tree setup time increases the overall multicast latency. Recursive Partitioning Multicast (RPM) [80], does not build a tree structure as in VCTM, but rather encodes the destination nodes' positions in the packet header. RPM uses a recursive hop by hop partitioning method, such that it replicates at a certain node. The replicated packets update the destination list in the header and generate a new network partitioning based on the current node. It outperforms the VCTM approach as setup requests are not required, but requires complex route computation logic. Path-based multicast routing ensures a route is determined whereby the flit traverses along a path to each destination sequentially. Implementation of such an adaptive routing algorithm, requires all turns to be allowed and virtual channels for deadlock avoidance.

The tree-based broadcasting mechanism in [51], mentioned as SMART-B throughout this chapter, broadcasts packets by creating shared or private virtual trees on a physical mesh network. By dynamically generating multi-hop bypass paths, this work aims to achieve a broadcast in a single cycle per dimension. Shared virtual trees result in contention among multiple broadcasts for the shared links. Private virtual trees eliminates the contention between broadcasts, by arbitrating for exclusive access to these broadcast links. A broadcast flit redirects to a corner node and awaits for access to the separate dedicated broadcast links. The buffering at the corner nodes allows for this mechanism to be feasible, however for bufferless networks the redirection latency would be gravely affected. SMART-B is the state-of-the-art buffered broadcast mechanism.

6.2.2 Bufferless Networks and Coherent Ring Architectures

Multicasting over networks, void of buffers, presents unique challenges with regard to flow control and guaranteed delivery to all destinations. Bufferless multicasting is rarely mentioned for on-chip mesh networks. In [32], multicasting is addressed by migrating the concepts of path-based and recursive partitioning to bufferless networks. The path-based bufferless multicast routes the multicast packet to each destination along a non-deterministic path, where the closest destination is selected to route to from the current router. However, the destination may vary as the flit is deflected, and does not guarantee flit delivery to all destinations. Recursive partitioning is similar to RPM, where the flit is replicated within the network if output ports are available.

Several chip prototypes and commercial multicore processors utilize ring interconnects: IBM Cell [21], Intel Larrabee [70], Intel Sandy Bridge [82], due to its simplicity in comparison to packet-switched networks and exploitable ordering for coherence. Numerous work on physical and embedded ring topologies [13, 72] are explored for medium range systems, but the ring is not totally ordered and may lead to retries.

A dedicated ordering point on the ring creates a global order with bounded latency, but hits performance due to the additional latency to reach the ordering point. Ring-Order [60] guarantees ordering and avoids retries by intercepting requests and data messages on the ring, and enforces transactions are completed in ring position order. Although inspired by token coherence, Ring-Order does not require persistent requests for forward progress. However, it requires the data to follow the ring path as well, increasing the network latency. Uncorq [73] broadcasts a snoop request to all cores followed by a response message on a logical ring network to collect the responses from all cores. This enforces a serialization of requests to the same cache line, but does not enforce sequential consistency or global ordering of all requests. Although read requests do not wait for the response messages to return, the write requests have to wait, with the waiting delay scaling linearly with core count, like physical rings.

6.3 Overview

SC^{2}EPTON (Snoopy Coherent, Single-Cycle Express Paths and self-Throttling Ordered Network) is a bufferless NoC architecture with efficient broadcast and unicast communication to support broadcast-based snoopy protocols. We leverage clockless repeaters [19] to ensure efficient, low-latency communication for all message types, with multi-hop traversals. Unicast messages are handled as in the SCEPTER architecture, with local starvation-awareness output port selection, and opportunistic bypassing along idle express paths. Forking broadcast messages at the NIC would only exacerbate the number of deflections and overall network congestion. Without the relief gained from buffering in the routers, flits continually deflect, consume additional link and crossbar power, and misroute if failed to access the ejection links.

Similar to the SCORPIO system, each node in the system consists of a main network router, notification router, and network interface controller. The SCORPIO main network is capable of handling broadcast requests on the GO-REQ virtual network and unicast requests on the UORESP and P2P virtual networks. The SCEPTER bufferless router architecture is capable of supporting high throughput unicast requests. For broadcasting, we tackle the problem of reduced network bandwidth and ejection link constraints, by maintaining synchronized time intervals, mentioned throughout as time-windows to achieve distributed time-division for un-contended access.

6.3.1 Multiplexed Non-Contending Broadcasts

The ring topology is quite simple - all routers are connected together in a loop, and traffic injected into the ring traverses all nodes until it reaches the destination(s). Since the topology allows for all nodes to be reached by hopping through the ring, complex routing is not required for broadcasting. With proper ring arbitration and access logic, in-ring buffering is not necessary. The ring latency is equal to N cycles, where N is the number of nodes, or $N/2$ if bidirectional links are used. In SC^{2}EPTON, each broadcast coherent request generated waits in the local NIC until the time window in which it is allowed to inject. The source IDs (SID), allowed to access the ring changes each time-window, such that all the sources are given a chance to inject a flit into the network. The clockless repeated links are utilized to enable multi-hop traversal in a single cycle. Thus, for a N node network, the ring latency for a broadcast is N/HPC_{max} cycles, where HPC_{max} is the maximum number of hops traversed in a cycle.

Generally, TDM-based arbitration in networks is performed at the sources or individual nodes as the requests arbitrate for access to the virtual channels, links, or switches. However, in a bufferless network, the latency effects of failed access to desired output links degrades performance as such flits are deflected in potentially non-progressive directions. The TDM-based mechanism must guarantee an end-to-end, contention-free path for efficient broadcast communication. However, even with clockless repeated links, the flit traversal reaches a maximum of 13 hops in a cycle at a clock frequency of 1GHz. [19] The key idea of the snake bufferless broadcast (SB^2) mechanism stems from the observations on link contention and ring arbitration:

1. **TDM-based Distributed Arbitration.** Synchronized time-windows can be used to perform distributed ring arbitration without the need of a centralized arbiter. Notifications enhance this TDM mechanism to ensure the network does not remain idle as would be the case with a static, vanilla TDM injection assignment. Instead, notifications enable dynamic allocation of the network, adapting to actual network load.

2. **Contention-Free Communication.** To eliminate contention on ejection links, more than one broadcast flit is disallowed from arriving at a node, at the same time. Throughput

suffers, however without single cycle full ring traversals, many ejection links are idle as the flit traverses through the ring over multiple cycles. Properly scheduling simultaneous flits from source IDs that do not contend for ejection links each time-window can ensure higher throughput and fully utilized links.

The SB^2 network is a ring that snakes through the SCEPTER mesh network, encountering every node on its path. At the beginning of a time-window, all source nodes with a flit waiting for network injection, determine locally the allowed injection SIDs for this interval. If allowed, the source proceeds with the flit injection into the ring. The flit enters the router, forks the flit to be sent to the local NIC, and proceeds to the next router along the multi-hop bypass path. The flit continues until it reaches HPC_{max} hops and is latched at that router. The enable bypassing signals are set without a reservation request as each node is aware of the allowed SIDs this time-window and whether it should allow a bypass or latch the incoming flit.

6.3.2 Walkthrough Example

More than one broadcast flit is able to simultaneously utilize the ring, provided they do not contend for links. For instance, in a 16 node mesh NoC, the virtual ring interconnects nodes 0->1->2->3->7->6->5->9->10->11->15->14->13->12->8->4->0. If source node 0 is able to inject, it it able to traverse up to HPC_{max} hops away before being latched. With a HPC_{max} of 4, the flit from $SID = 0$ can reach up to node 7, where it has to be latched. The rest of the ring is idle and can accommodate additional flits. To maximize simultaneous flit injections, the ring is demarcated into regions of HPC_{max} hops each. Thus, flits from nodes 7, 10, and 13 are allowed to inject at the same time as node 0.

Figure 6-2 depicts a diagram of the bufferless broadcast NoC, utilizing the notification network (Chapter 3) with enhanced clockless links for shorter time windows. At varied times before T0, cores 0, 2, 7, 8, 13, and 15 observe cache misses and send broadcast coherent requests to the NIC for network injection. The NICs encapsulate these requests into single flit packets and generates a separate notification bit-vector, to be broadcast to all nodes.

At T0, notification messages arrive at all nodes via the notification network. The merged notification is as shown, an asserted bit in the field corresponding to the source id of the node,

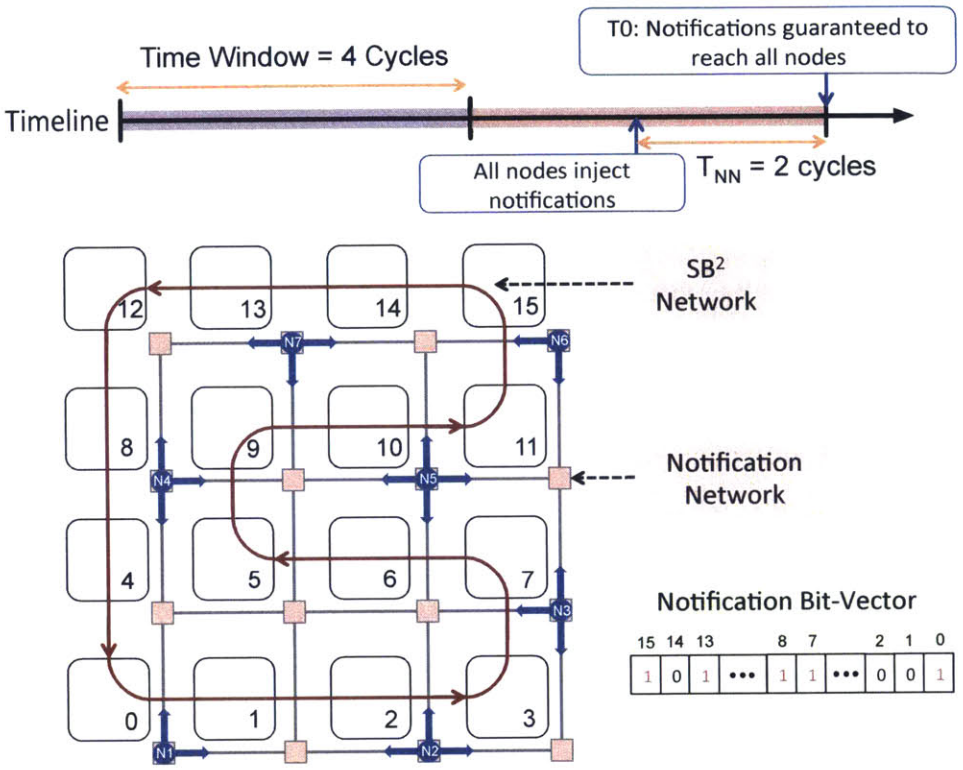


Figure 6-2: Walkthrough Example of TDM-based Single-Cycle Bufferless Broadcast - Notification

requesting broadcast network access. Each node processes the notifications and performs a local, but consistent determination of the SIDs that win ring access this time-window (TW). With the notification received, each node locally decides to allow nodes $\{0, 7, 10, 13\}$ to simultaneously inject this time-window. Nodes 0, 7, and 13 inject requests M1, M2, and M3, respectively. During the time-window, the injected flits traverse the ring, taking HPC_{max} hops per cycle. In the 16 node example, HPC_{max} is equal to 4, resulting in time-window of 4 cycles, which is sufficient for up to four injected flits to reach all destinations. Each cycle, the flits are latched at the next router, HPC_{max} hops away. Figure 6-3 shows that after one cycle, M1 is latched at node 7, M2 is latched at node 10, and M3 is latched at node 0. This unidirectional snake ring ensures exclusive access to ejection links, simultaneously allowing multiple flits to traverse demarcated sections of the ring. Non-contending sets of nodes are [$\{0, 7, 10, 13\}$, $\{1, 6, 11, 12\}$, $\{2, 5, 8, 15\}$, $\{3, 4, 9, 14\}$].

When switching to the next time window at T4, the notifications indicate cores 2, 8, and 15 still need to inject coherent requests into the network. Switching through the non-contending sets, the next applicable set with sources containing valid notifications is $\{2, 5, 8, 15\}$. The three injected

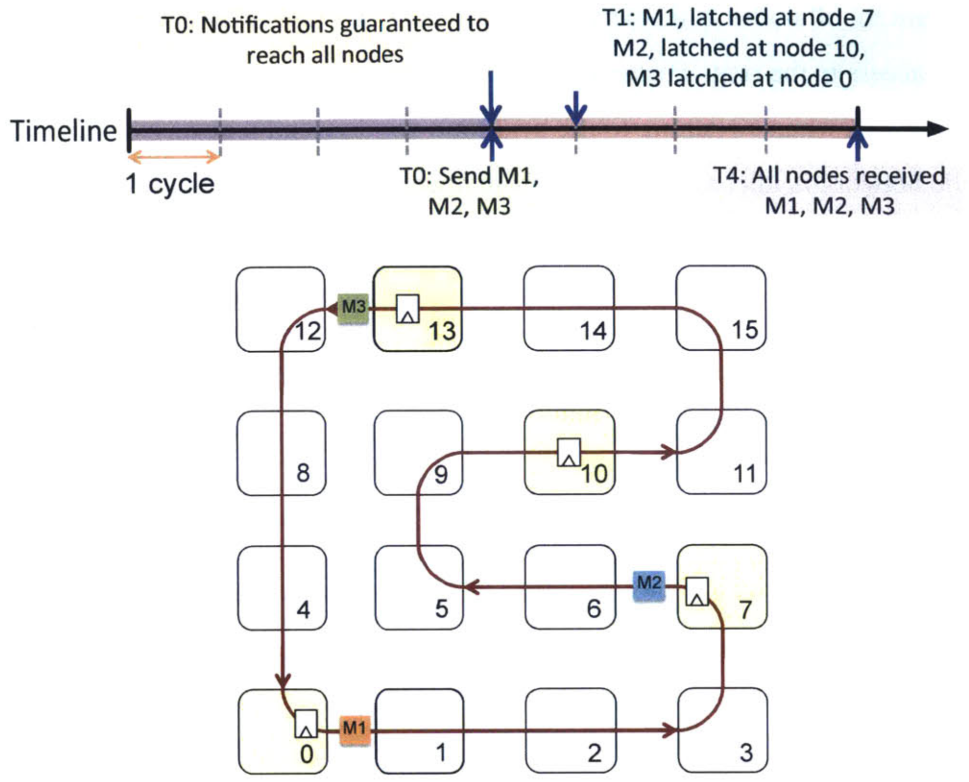

T0: Notifications guaranteed to reach all nodes
T1: M1, latched at node 7
M2, latched at node 10,
M3 latched at node 0
Timeline
1 cycle
T0: Send M1, M2, M3
T4: All nodes received M1, M2, M3
M1
M2
M3
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Figure 6-3: Walkthrough Example of TDM-based Single-Cycle Bufferless Broadcast - Distributed Ring Arbitration and Traversal

flits reach all destinations within the next 4 cycles. Data responses are sent on the SCEPTER mesh NoC to the requesters as only unicast message communication is necessary.

6.4 Single-cycle Bufferless Broadcast (SB^2) Network

The SB^2 network is part of the large SC^{2}EPTON architecture containing a bufferless unicast network (SCEPTER) and a distributed ordering policy using notifications (SCORPIO). Combined with this bufferless broadcast network, the fully bufferless NoC is capable of high performance coherent communication, while eliminating the area/power overheads of directory storage and router buffers.

Considering a static TDM ring access assignment, every cycle, up to N/HPC_{max} flits traverse the network without contention. The static TDM assignment cycles through all non-contending node sets, covering all the potential request sources in the network. For instance, a 16-node NoC has the following non-contending sets of nodes: [$\{0, 7, 10, 13\}$, $\{1, 6, 11, 12\}$, $\{2, 5, 8, 15\}$, $\{3, 4, 9, 14\}$], which account for all nodes in the network. During the first time-window, the first set, {0,7,10,13} are granted access to the ring. The second time-window, the ring allows {1,6,11,12} to inject. This process continues when switching to the next time-window and repeats over and over. However, assume the network is idle, and a request from node 9 needs access to the SB^2 ring. In this case, the worst case latency is when the node waits for three time-windows, during which the network is idle, and is only granted access in the fourth time-window. Although the SB^2 network latency is significantly reduced with clockless repeated links, the worst case latency prompts concerns especially when the network is remaining idle.

6.4.1 Dynamic and Distributed Ring Arbitration

To solve this problem, we use notifications to achieve a dynamic ring arbitration scheme, by broadcasting the network access requesters to all nodes. Utilizing synchronized time-windows, each node locally determines which nodes are granted ring access. Thus, in 16-node network in the walkthrough example, each node cycles through the non-contending sets and determines if the notification message sources match any nodes in the set. If not, none of the requesters are in this non-contending set of nodes, and the next non-contending set is checked. As each node performs this check, all will arrive at the same conclusion on the nodes allowed to inject into the ring this time-window. This is guaranteed as the notifications are received by all nodes within a fixed latency.

The notifications propagate to all nodes on a separate, bufferless network that guarantees a fixed latency bound by merging notifications upon contention. The notification network, described in Chapter 3, latches notification messages at each router in the network, incurring a fixed latency bound of $2k$ in a k$\times$k mesh NoC. Leveraging the clockless repeated links [19], once again, we are able to obtain the low latency benefits, thereby reducing the notification broadcast latency.

To achieve multi-hop bypassing of routers, the bypass path is preset using a reservation requests, as described in Chapter 5. Since the notification network is a static broadcast network, a dynamic reservation mechanism is not required. Rather, the bypass path is enabled for straight through traffic, and notifications are latched at turns. A notification arriving from the *East* is able to bypass through to the *West* direction as the bypass is statically enabled. At each node , the notification is bitwise-ORed with other notifications with the same output direction. At a "turn", the notification is latched and sent out the following cycle. Considering an HPC_{max} less than the maximum

Manhattan distance between source and destination, a 2-cycle notification network broadcast latency is generally achieved. For larger networks or lower HPC_{max} values, the notifications must be latched more often.

6.4.2 SB^2 Router Microarchitecture

The ring topology is a simple network that is very useful for broadcast communication as each node is traversed along the ring. With only two links per node, this degree 2 network router is very simple and consumes minimal area, especially when realized without virtual channels and completely bufferless. Figure 6-4 shows the microarchitecture of the SB^2 router. In addition to the single input and output link of the ring, the NIC input link for injection into the network, and NIC output link for ejection from the network is shown. When a flit arrives at the router, it either bypasses through to the asynchronous repeaters shown, or buffers in the latch for one cycle. Three control signals manage the NIC injection and bypassing setup: Inj_{EN}, Byp_{EN}, R_{SEL}. When Inj_{EN} is asserted, the NIC is able to send the flit into the network and to the multiplexers where the flit is forked to the output links. Similarly, the incoming flit from the *West* port is able to bypass when Byp_{EN} is asserted and the multiplexer logic forks the flit to the local NIC and the *East* output direction. R_{SEL} must be set to allow either the flit from the NIC or the *West* input to be sent through to continue the

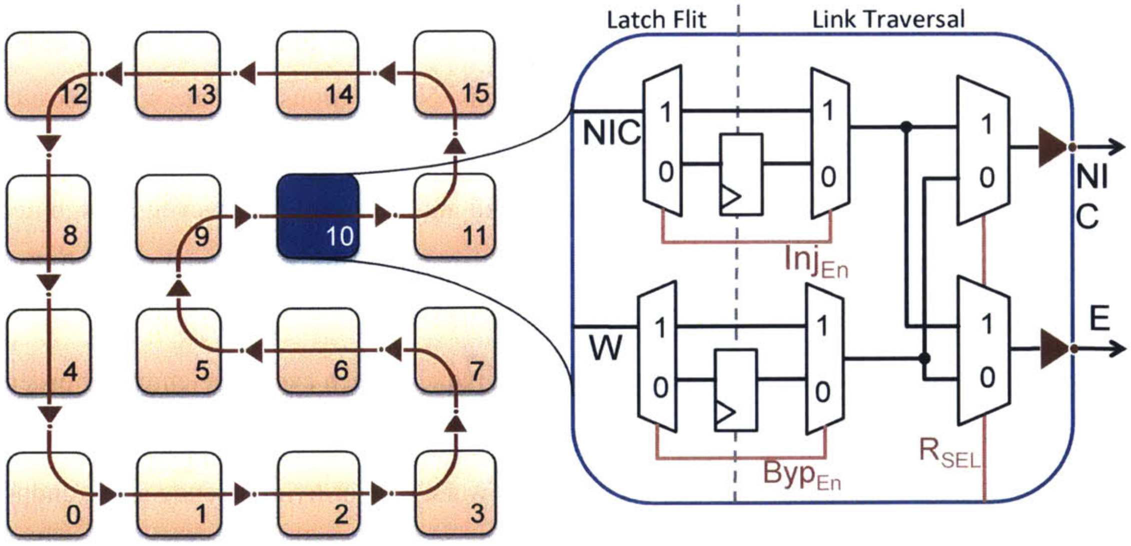


Figure 6-4: SB^2 Snake Ring and Router Microarchitecture

broadcast. Hence, the signals are assigned to eliminate output link contention.

Let's revisit the 16 node walkthrough example, once again, to illustrate the contention-free behavior. In a distributed manner all the nodes are aware of the injecting sources. At this time, the control signals for the SB^2 router are set. When nodes {0,7,10,13} are allowed to inject, every node is aware whether its own node or another node is injecting. Thus, nodes {0,7,10,13} set Inj_{EN} to 1 and the R_{SEL} to 1 such that the NIC's flit enters the network. The Byp_{EN} is set to 0, to eliminate output link contention and ensure the flit is latched to be sent the following cycle. All other nodes are aware that they are not allowed to inject this time window and enable the bypass path for flits on the ring. Thus, setting Byp_{EN} to 1, Inj_{EN} to 0, and R_{SEL} to 0.

6.4.3 Global Request Ordering

The refreshingly simple broadcast network is effective at delivering broadcasts to all nodes, in a contention-free manner, and with low latency. However, to support snoopy coherence and sequential consistency, every node must consistently order requests such that a global order is achieved.

In the SCORPIO NoC, each time window, notifications inform of requests from certain source nodes that are expected to arrive via the main network. Following consistent ordering rules, every node prioritizes the source IDs and processes requests in the priority order. The worst case is when flit arrives prior to higher priority flits and is prioritized last. Thus, the flit must reside in the network interface buffers until all other flits are received and processed. With the use of notifications, the SCORPIO NoC eliminates the worst case for low loads. Notifications indicate which source IDs have valid requests arriving; Allowing a seemingly low priority flit to be processed sooner as the NIC is aware that flits from higher priority sources will not be arriving. However, at high network loads, the worst case is still present and the ordering latencies can be detrimental to the full system application performance.

By constraining the potential number of broadcast requests each cycle, the endpoint ordering is very efficient. Each time-window, up to a maximum of four broadcast requests can arrive in a 16 node system, with HPC_{max} of 4. In this scenario, the worst case ordering latency at the endpoint is 6 cycles for the SB^2 network: 3 cycles awaiting the highest priority request, and 3 cycles to send the three ahead requests, one at a time, up through the NIC and to the cache controller.

Point-to-Point Ordering

Sequential consistency requires that program order is maintained for individual processors. In addition to enforcing a global order, requests from the same source need to be ordered as well, referred to as point-to-point ordering. The dynamic ring access allows one flit per node to be sent in N cycles, where N is the number of nodes. The NIC processes requests and places them into a FIFO, to await network access. Thus the request issue order is maintained as the flits are scheduled in-order for network access, and multiple flits from the same source are not allowed simultaneously.

Finite Endpoint Buffering

NIC buffers hold flits until they are ordered, depacketized, and sent to the cache controller. A 16 node network has a 4 cycle time-window, and is able to handle up to four flits within that time. At the endpoint, worst case occurs when 3 flits arrive prior to the highest priority request. By the end of the time-window, the fourth flit arrives and is sent to the cache controller the following cycle. Worst case, the destinations must have three buffers to accommodate the waiting flits. These destination buffers must be appropriately sized for performance and correctness reasons. A maximum of N/HPC_{max} flits are handled per time-window. At each bufferless router, the flit must exit either the same or following cycle, based on the bypass and latch control signals. If some NICs along the ring are unable to accept the flit, where the number of buffers is less than N/HPC_{max}, the broadcast will fail.

SCORPIO has a worst case sizing of 2 buffers in each NIC, irrespective of the number of nodes, for correctness, but performance goals required at least 4 buffers for the ordered vnet. The router buffers account for buffer backpressure due to the NIC buffers, reducing any performance impact. Due to the lack of router buffers in SB^2, buffer backpressure at routers cannot be leveraged to alleviate fully occupied endpoint buffers and worst case buffer sizing in the NICs must be enforced.

6.5 Architecture Analysis

Modeled System. For synthetic and full system architectural simulations, we use the gem5 [15] cycle-accurate simulator with the GARNET [10] network model. The SC^{2}EPTON and baseline

Table 6.2: System Parameters for Evaluations

Core	x86 in-order
L1 cache	32 KB I/D
L2 cache	1 MB
NoC Topology	6×6 (full system), 8×8 mesh (synthetic)
HPC_{max}	8
Notification Window	2
Virtual Channels	12

architectural parameters are depicted in Table 6.2 for the full system simulations.

Baselines. Broadcast and multicast support in on-chip buffered networks are usually achieved by converting a single multicast into multiple unicasts, establishing a multicast tree in the network with forking within the routers, or routing through a path that encounters the multicast destinations sequentially. SCORPIO [28], an ordered mesh NoC, utilizes a tree-based broadcast support within the routers. Instead of sending the same request as multiple unicasts, it allows request to fork through multiple router output ports in the same cycle, along a deterministic XY tree. SMART-B [51], leverages the single-cycle multi-hop traversals in [50] for private virtual broadcast trees in a mesh NoC.

Evaluation Methodology. We evaluate the performance of the SB^2 bufferless broadcast network with synthetic traffic, and compare with the SCORPIO and SMART-B baselines. For fair comparison, the notification network enhanced with asynchronous repeated links are incorporated in the SCORPIO architecture as well, resulting in a shorter time window. Thus, the wait time until another flit is able to inject from the same source is reduced tremendously, from a maximum of 16 cycles to 2 cycles for a 64-node network. Insight and conclusions obtained from the synthetic results, are used to specify the architectural parameters of the SB^2 network within the SC^2EPTON architecture.

We evaluate the full system performance for the SC^2EPTON network compared to the baseline buffered networks, supporting snoopy (SCORPIO) or directory-based (SMART-B) coherence. Since SC^2EPTON and SCORPIO support global coherent request ordering, both use the MOSI broadcast protocol, explained in-depth in Chapter 4, SMART-B network uses the AMD Hypertransport protocol to maintain coherence as the directory is the ordering point. AMD Hypertransport uses the directory as an ordering point, and thus can function with just a small directory cache, but sacrifices by relying on broadcasts. The SMART-B network handles these broadcasts, while SMART [50] handles the unicast communication. The combined network is referred to as SMART-Full throughout this section. It is a state-of-the-art buffered network that efficiently handles unicasts and broadcasts.

6.5.1 SB^2 Network-Level Performance Analysis

We evaluate the latency and throughput of the SB^2 network for 64 nodes and compare its performance to the SCORPIO [28] and SMART-B [51] networks, each with 12 VCs per input port. Both SMART-B and SB^2 are evaluated with an HPC_{max} of 8, unless mentioned otherwise. Dynamic TDM ring arbitration, with a bufferless, contention-free, fast notification network, yields improved performance, as we discussed qualitatively earlier. Here, we evaluate its performance effects through detailed cycle-accurate simulations for further insight.

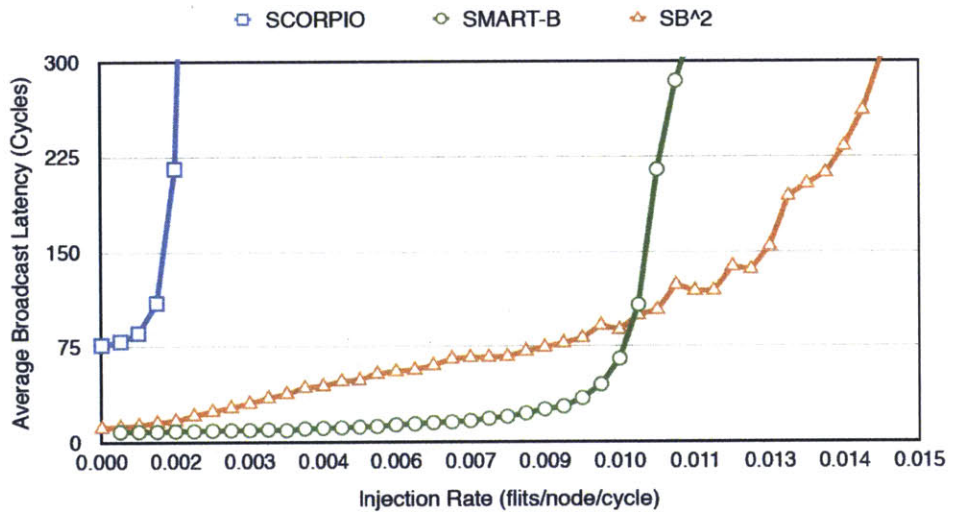


Figure 6-5: Comparison of Broadcast Latency with Buffered Broadcast Optimized Networks

Broadcast Latency and Throughput for Uniform Random Traffic.

Figure 6-5 is a plot of the broadcast latency, i.e. time for broadcast to reach all destinations, as a function of the injection rate for an 8×8 network and when all nodes have equal probability of injecting a flit. It includes both the network and source queueing delay. The ordering latency at the destination for SCORPIO and SB^2, and directory serialization latency for SMART-B, is not considered here, so that the network performance alone is extracted for comparison.

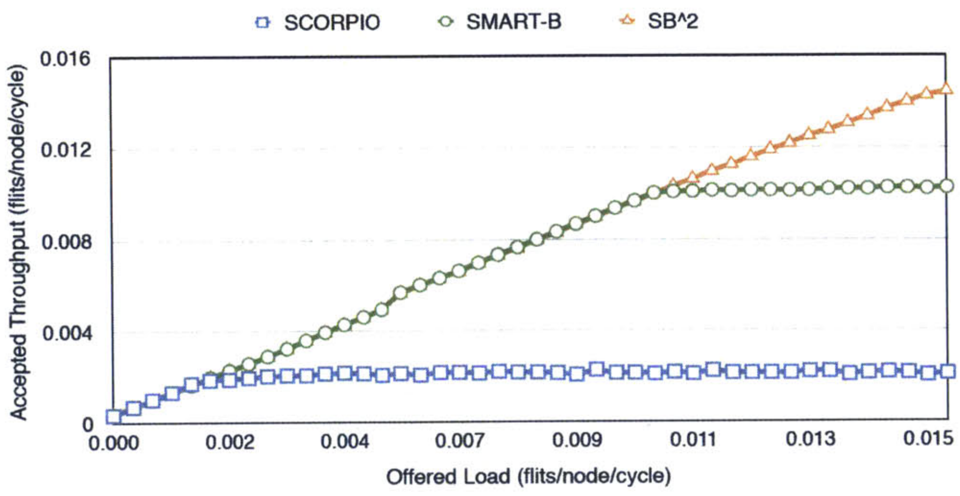


Figure 6-6: Throughput for Broadcast Communication in 8×8 Network

SCORPIO does not have dedicated broadcast links or separate physical network to handle broadcast requests. Rather each broadcast contends with flits from other message classes for access to the same physical links and switches. SMART-B, and SB^2, have separate dedicated links for achieving broadcast performance improvement, hence contention with unicast flits is avoided at both the inter-router links and ejection links. SMART-B buffers flits at corner routers, until a private virtual tree is established, and the broadcast completes in at most two cycles. To eliminate unfair broadcast performance comparison, here we evaluate the broadcast latency only and send unicast traffic on a separate, ideal 1-cycle network for all NoCs. Even with broadcast support within the SCORPIO routers to fork flits at each hop, the throughput performance lags SMART-B. The broadcast latency shown accounts for the source queueing latency, which includes the time until a message is able to inject into the network. Thus, over-provisioning the SCORPIO routers with buffers does not help throughput when the injection policy restricts one flit to be sent per source per time-window. SB^2 begins with an 11 cycle latency, at the zero load point, and gradually increases as the network injection rate is increased. Using the derivation of the ideal broadcast latency and throughput for a 8×8 mesh NoC in Table 6.1, the ideal throughput is $R/(k^2 * R) = 1/k^2 = 0.015$ flits/node/cycle, and ideal latency is 8 cycles. The SMART-B network saturates at 66% of the ideal throughput limit, where nearly all the buffers in the network are saturated and queueing delay is constant. However, the SB^2's saturation throughput cannot be as easily identified. We plot the accepted throughput as a function of the offered load in Figure 6-6, for a 8×8 network. SB^2 is provides a throughput equal to the offered network load.

The NICs are equipped with at least the minimum number of buffers to avoid the case where the maximum number of injection sources cannot all proceed. Hence, SB^2's network interfaces do not add backpressure into the network, and saturation doesn't occur.

Dynamic TDM Ring Arbitration with Uniform Random Traffic.

The static TDM ring arbitration mechanism, TDM-Static, uses the same ring network as SB^2 but resorts to a static assignment of ring access each time-window. It repeatedly cycles through the sets of non-contending sources, thereby covering all nodes in the network. Each node must wait for its time slot to inject, resulting in a worst case wait time of N/HPC_{max} cycles, where N is the number of nodes. For low loads, time slots are cycled through, however hardly few of the nodes allowed to inject, actually do so. Figure 6-7 depicts the broadcast latency for the static TDM approach and dynamic notification-based arbitration mechanism, on uniform random traffic for 64 nodes. The notifications reduce the latency by 25 cycles at approximately the zero load point. After about 0.006 flits/node/cycle, each time slot contains at least one valid request to inject for the uniform random traffic profile. Thus, due to the uniform random source selection, no time slots are idle for higher injection rates and the same results are observed for the static and notification-based approaches. For application workloads, the dynamic ring arbitration is even more beneficial, especially in the event of bursty traffic originating from a single or few sources. Global ordering can be maintained with

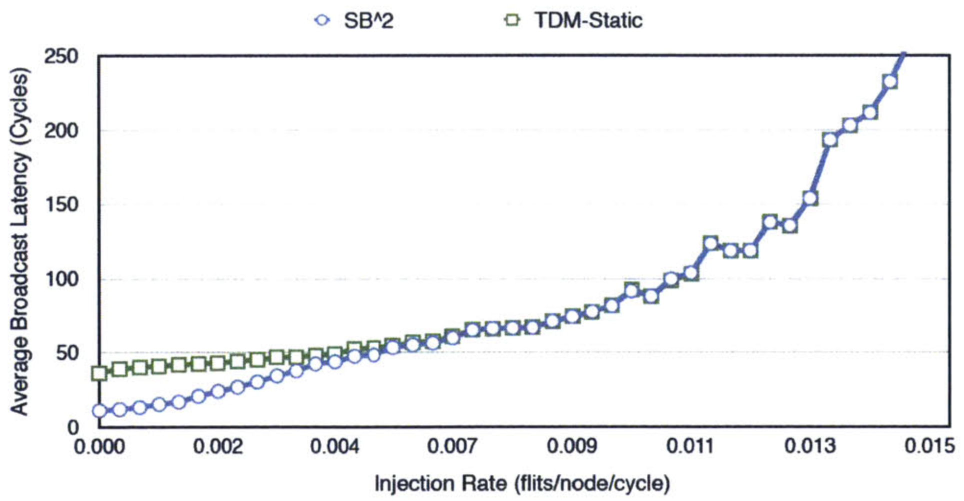


Figure 6-7: Broadcast Latency with Static and Dynamic TDM Arbitration for 64 Nodes

both approaches, as each node locally orders requests according to *consistent* rules. Notifications reduce the ordering latency at the endpoints in the event the time windows are not fully utilized by injecting sources.

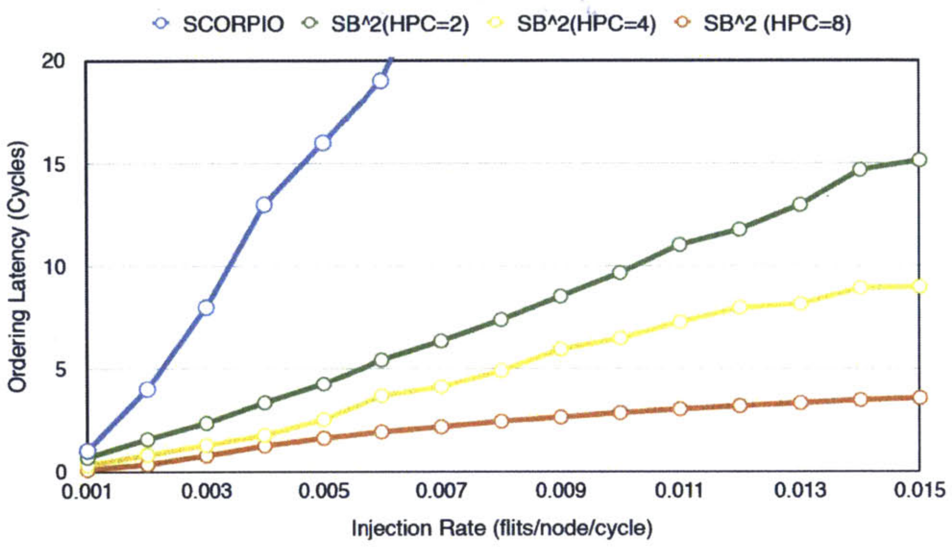


Figure 6-8: Comparison of Ordering Latency of SB^2 and SCORPIO

Ordering Latency.

Optimizing the endpoint ordering latency is crucial for improved performance of snoopy coherence in comparison to directory-based coherence. The ordering latency should be much less than the indirection latency for the approach to be feasible. SCORPIO outperforms directory baselines because the ordering latency is less than 50% of the indirection plus serialization latency. Figure 6-8 depicts the ordering latency of SCORPIO in comparison to SB^2, with respect to the injection rate. Due to fewer requests being ordered each time window, and fixed network latencies after network injection, SB^2's ordering latency is substantially lower than SCORPIO's. As HPC_{max} is reduced, the ordering latency increases. This is due to the property where multiple simultaneous sources are allowed to inject. With shorter multi-hop traversals, there are more non-overlapping segments of the ring available. Thus, for 64 nodes and a HPC_{max} of 4, each time-window allows up to 16 flits to enter and route through the ring.

With the reserved virtual channel in SCORPIO's main network, and each hop incurring 1 cycle with lookahead bypassing, the highest priority flit arrives within 16 cycles for the 64-node network. However, the SMART-B network with directory-based coherence has a lower indirection latency due to the optimization of unicast messages with dynamically preset multi-hop traversals. The ordering latency of SCORPIO prohibits performance improvement over such state-of-the-art buffered networks with minimal indirection latency.

Latency and Throughput Sensitivity to Hops-Per-Cycle.

As HPC_{max} is reduced, the number of simultaneous injection sources increases, as well as the network latency. For 64 nodes, a HPC_{max} of 2 allows 32 nodes to inject simultaneously, thus a maximum of 32 flits reach all nodes within 32 cycles. Higher hops per cycle results in lower network latency, smaller time windows, and reduced waiting time for a time slot. Without global ordering support, it is evident that the throughput is the similar as HPC_{max} is varied, with a maximum of 64 flits arriving at all destinations within 64 cycles. However, ordering of requests is required at the destinations for snoopy coherence and sequential consistency support, which imposes a waiting delay at the endpoints as well, reducing network throughput. Figure 6-9 shows the waiting latency at the source, and throughput as HPC_{max} is varied. The ordering latency variation is depicted in Figure 6-8. For low injection rates, the higher the HPC, the lower the wait time, since more idle time windows occur.

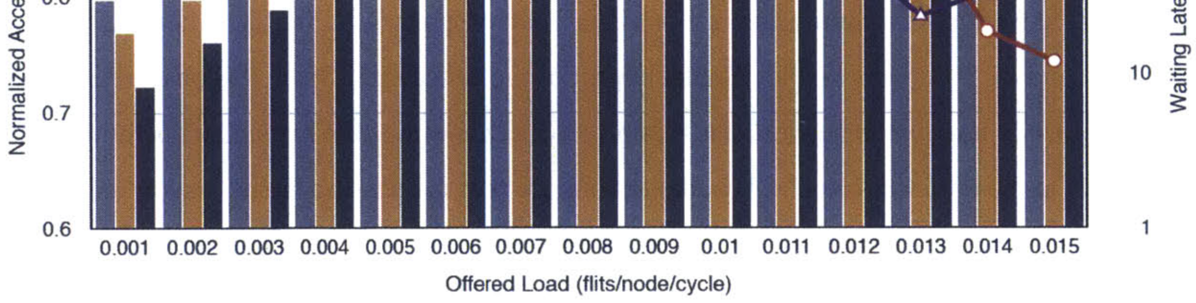


Figure 6-9: Sensitivity to HPC_{max}

6.5.2 SC^{2}EPTON Analysis and Discussion

SC^{2}EPTON consists of a bufferless mesh architecture and embedded snake ring to communicate coherent responses and requests, respectively. The network-level analysis of the bufferless response network, SCEPTER, and embedded ring request network, SB^2, displayed high-performance in comparison to baseline buffered networks.

Application Performance

Full system application performance on a 36 node system is performed to compare the SC^{2}EPTON bufferless network with the snoopy coherent SCORPIO network, and directory-based SMART-Full network. SC^{2}EPTON divides messages to perform bufferless broadcast over SB^2 and bufferless unicast over SCEPTER. Similarly, we model the SMART-Full network to have separate unicast and broadcast networks.

Figure 6-10, shows the normalized application runtime for SPLASH-2 and PARSEC benchmarks simulated on gem5. SC^{2}EPTON achieves on average a 9% lower application runtime than SCORPIO, and 13.6% higher runtime than the SMART-Full network. *Swaptions* generates lower than average L2 cache miss rates and reduced network traffic. As a result, the benefits of SMART-Full and SC^{2}EPTON over SCORPIO only reduce the runtime by 10% and 6%, respectively. Subject to the communication pattern, opportunistic bypassing may or may not be beneficial. In the *water-spatial* application, 26% of opportunistic bypass paths are not taken. Thus, for unicast data responses, the deflections cause additional latency penalty. Coupled with the performance impact of the bufferless broadcasting, the results show a wider performance gap between SMART-Full and SC^{2}EPTON.

SC^{2}EPTON's performance gains over SCORPIO, despite the lack of buffers, is due to efficient broadcast and unicast communication. For unicast data responses, SC^{2}EPTON benefits from path diversity and express paths to alleviate the impact of the deflections. Since it is observed that these applications do not tax the network and exhibit low network injection rates, this operating point is beneficial to the bufferless NoC. Thus, deflections are few, the per-hop latency is low as the router pipeline is shorter, and express paths are set up and traversed by the reserving flit.

More so, the benefits to the broadcasts are evidents from the evaluation of the broadcast latency and ordering latency of SB^2 (embedded in SC^{2}EPTON) and SCORPIO. Even at low injection

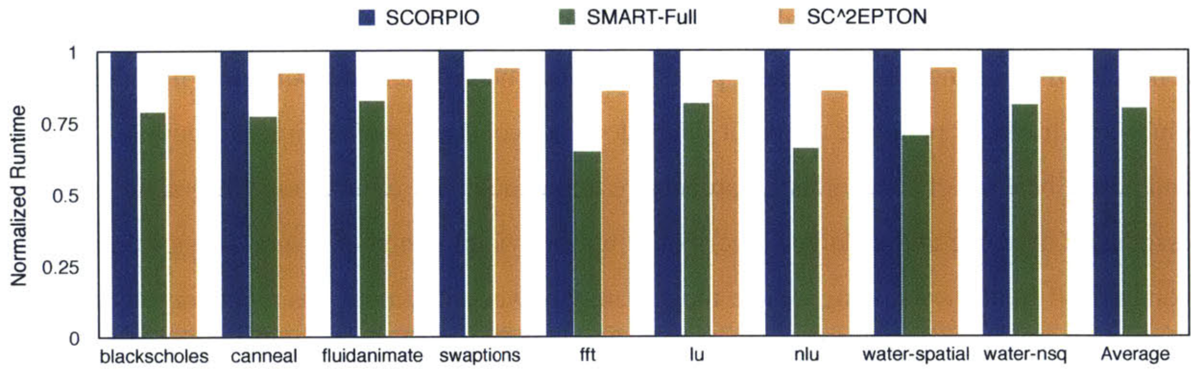


Figure 6-10: Normalized Performance of Full System PARSEC and SPLASH-2 Applications

rates, the ordering latency is reduced in comparison to SCORPIO. For SCORPIO, we bound the notification network latency but the main network's performance isn't bounded and depends on the network traffic, contention, and priority. For deadlock avoidance, the reserved VC ensures the highest priority flit is granted a single cycle per hop lookahead bypass path.

Area and Power

The energy breakdown of the components in the SCORPIO and SCEPTER NoCs were discussed in detail in prior chapters. The SB^2 ring network has minimal overhead as each router's data and control path is very simple. Effectively, the buffers at the network interface increase slightly to accommodate for the maximum number of requests per time window, while the router buffers are completely eliminated. Using the IBM 45 nm SOI technology, the SC^2EPTON RTL synthesis with Synopsis Design Compiler, we determine the post synthesis area and power. The SC^2EPTON NoC reduces the combined router and NIC area by 39%, and power by 36%, in comparison to the SCORPIO NoC.

Discussion

SC^2EPTON demonstrates that such bufferless networks can enable high performing snoopy coherence at low cost. Especially without the complexity that ordering usually entails, in the form of timestamps, tokens, etc. Bufferless networks are often targeted for low to medium loads, since the performance degrades for higher loads. SC^2EPTON achieves high performance bufferless communication for coherent requests and responses, while maintaining global ordering support for snoopy coherence and sequential consistency. The switch allocators for bufferless networks must schedule an incoming flit to an output port. Since the critical path of the bufferless mesh router is the multi-hop traversal path, the allocator delay does not constrain the maximum frequency. The area and power benefits of a bufferless approach are tremendous. By eliminating buffers, we remove the associated dynamic and leakage power consumption, yielding higher power savings than dynamic voltage and frequency scaling, where the dynamic power consumption is alone targeted.

CHAPTER 7

CONCLUSION

Since the end of the golden age of Dennard scaling, achieving uniprocessor performance improvements per technology generation is not as straightforward, especially within tight area and power budgets. The limitations commonly referred to as a brick wall (ILP+Memory+Power), turned the attention to multicore processors as a saving grace, with the view that processors are the future transistors. However, to reach the stage where many processors are easily integrated, the processors must (1) communicate efficiently, (2) ensure coherence, and (3) remain within the area/power constraints. Processor architectures are quite diverse, with varying power and area budgets, allowing for a good selection when integrating many processors on-chip. The challenges lie in the interconnect, where the three points mentioned translate to low-latency and high-throughput communication, with cache coherence support, and low power/area cost. As hardware cache coherence provides greater performance potential than the software counterparts, it is often optimized and focused on in research communities, in an attempt to scale hardware cache coherence as far as possible.

The design of the on-chip network plays a crucial role as maintaining cache coherence trades on the performance and area/power efficiency of the network. This dissertation outlines the three networks that target the challenges of supporting high performance cache coherence, at ultra-low overhead. The three-fold NoCs are combined into SC^{2}EPTON (Snoopy Coherent, Single-Cycle Express Path, self-Throttling, Ordered Network), which takes parallel architectures and on-chip networks beyond the conventional (buffered, directory-based) to SCORPIO's 36-core chip prototype (buffered, snoop-based), and towards low-power, intelligent interconnects (bufferless, snoop-based) to address the limits to scaling coherence.

7.1 Summary and Contributions

The dissertation steps through the development of the SCORPIO, SCEPTER, and SB^2 NoCs, and culminates in the SC^{2}EPTON bufferless NoC architecture for scalable many-core chips.

7.1.1 SCORPIO: Distributed Global Ordering

Directory-based cache coherence is the conventional choice for multicore processors, but incurs high storage overheads, limiting its scalability. Reducing storage results in inaccurate sharer tracking, thereby deteriorating application performance. While snoop-based cache coherence eliminates the storage overhead requirement, it intrinsically relies on ordered interconnects which do not scale. SCORPIO, a network-on-chip (NoC) architecture containing a separate fixed-latency contention-free network, supports snoopy coherence with a novel distributed global ordering mechanism for unordered mesh networks at low cost.

Key Idea: *Message delivery is decoupled from the ordering, allowing messages to arrive in any order and at any time, and still be correctly ordered.* Upon a cache miss request from the core, the network interface packetizes and generates two network request types. One is a broadcast message containing the address and sent to all cores, such that one may respond with the data. The other message is a notification, informing all cores to expect a request from this core. Since knowledge is imparted globally, each node orders messages locally while maintaining global order with consistent ordering rules and synchronized time windows. Each time window, the prioritization order changes for fairness reasons.

The architecture is designed to plug-and-play with existing multicore IP and with practicality, timing, area, and power as top concerns. The SCORPIO architecture is incorporated in an 11 mm-by-13mm chip prototype, fabricated in IBM 45nm SOI technology, comprising 36 Freescale e200 Power Architecture cores with private L1 and L2 caches interfacing with the NoC via ARM AMBA, along with two Cadence on-chip DDR2 controllers.

Insight: With notifications, ordering at the endpoints is performed based on the actual injected messages, rather than expecting the worst case conditions - all nodes have sent requests. This minimizes the ordering delay such that it is much less than the indirection latency plus serialization latency incurred in directory-based coherence. Full-system 36 and 64-core simulations on SPLASH-2 and PARSEC benchmarks show an average application runtime reduction of 24.1% and 12.9%, in comparison to distributed directory and AMD HyperTransport coherence protocols, respectively. The chip prototype achieves a post synthesis operating frequency of 1 GHz (833 MHz post-layout) with an estimated power of 28.8 W (768 mW per tile), while the network consumes only 10% of tile area and 19 % of tile power. While the 36-core SCORPIO chip is an academic chip design that can be better optimized, we learnt significantly through this exercise about the intricate interactions between processor, cache, interconnect and memory design, as well as the practical implementation overheads.

7.1.2 SCEPTER: High-Performance Bufferless Network

Although the downsides of directory-based coherence are averted, in SCORPIO, the network itself consumes a significant fraction of the total chip power, of which the router buffer power dominates. As higher performance is desired from the network, each router is packed with buffers, prompting area and power concerns. Eliminating router buffers while retaining high performance is ideal for future manycore systems. SCEPTER, a bufferless NoC architecture, achieves high performance unicast communication by setting up single-cycle virtual express paths dynamically, allowing deflected flits to traverse non-minimal paths with no latency penalty. By leveraging asynchronous repeated links, express paths are set up dynamically. However, in a bufferless router, a flit cannot be buffered to allow for bypassing flit from far away, to pass through this router the following cycle.

Key Idea: *Leverage single-cycle express paths to reduce the latency penalty of deflections in bufferless networks* SCEPTER intelligently prioritizes between flits currently in the router pipeline, flits bypassing from faraway, as well as flits waiting in the network interface to be injected. It adaptively routes flits in a livelock-free manner while maximizing opportunities to zoom along virtual express paths by opportunistically bypassing. For high network loads, it self-learns the throttle rate at each node, reducing starvation in the network. The bufferless network takes advantage of path diversity with deflection routing, and deliberate selection of non-minimal output directions when the flits are constantly ping ponging between nodes in heavily congested regions.

Insight: The chaotic environment of a bufferless network, where flits deflect and continually roam the network until it reaches the destination, is unpredictable as scheduled preset multi-hop traversals are wasted in the event the flit doesn't arrive. SCEPTER takes advantage of these idle multi-hop paths by allowing other flits to bypass the router if the crossbar switch has been preset, and the route would not deflect the flit farther from the destination. Utilizing these opportunistic bypass paths and output port selection, performance is greatly enhanced for bufferless networks. For a 64 and 256 node networks, we demonstrate 62% lower latency and $1.3\times$ higher throughput over a baseline bufferless NoC for synthetic traffic patterns on average; on par performance to a buffered mesh network with 6 flit buffers per input port in each router. Self-throttling with a hotspot traffic pattern, results in a 31.4% and 38.6% lower starvation rate for 64 and 256 nodes. Finally, full-system 36-core simulations of SCEPTER on SPLASH-2 and PARSEC benchmarks show an application runtime reduction of 19%, in comparison to a baseline bufferless NoC; and comparable performance to a single-cycle multi-hop buffered mesh network. This is achieved with a completely bufferless NoC, with SCEPTER leading to 36% area and 33% power savings vs. a state-of-the-art buffered NoC.

7.1.3 SB^2 Dynamic and Distributed Snake Ring Arbitration

For unicast communication, SCEPTER performs on-par with state-of-the-art buffered networks. However, snoopy coherence requires broadcast communication to be achieved with low latency, for overall performance gains. Broadcasts significantly congest the network links in a bufferless architecture: many deflections at each router, reduce multi-hop bypassing opportunities, high network latencies, and ejection links. SB^2, a bufferless broadcast NoC architecture, targets the broadcast contention effects, further exacerbated in bufferless interconnects, utilizing a ring that routes through all nodes. Arbitration for ring access is performed using synchronized time windows where each node is aware of the sources allowed to send requests into the network, for that time slot. This time-division-multiplexed method removes the need for a centralized arbiter, however for snoopy coherent communication, all requests need to be globally ordered.

Key Idea: *Dynamically determine the time-division-multiplexed access for the ring, allow simultaneous ring access for non-contending sources, and set router control signals, all locally at each node.* As the broadcast bandwidth is often limited by the ejection links, un-contended access ensures timely delivery of the broadcast request to all nodes. A distributed, dynamic time-division multiplexed mechanism is created to allow flits to simultaneously utilize the network and obtain un-contended access to ejection and output links each cycle, without a centralized arbiter. Each node locally determines the nodes granted network access, while maintaining a globally consistent arbitration decision.

Insight: The ring presents an ideal topology for bufferless broadcasting as the network topology ensures traversal of all nodes. However, the ring topology does not scale well as the latency increases with N, where N is the number of nodes. Using asynchronous repeated links, this latency is reduced which improves the performance of broadcasts. Together with the time-division-multiplexed arbitration, the SB^2 network does not saturate and continues to deliver flits at destinations at a rate that closely matches the offered load.

7.1.4 SC^{2}EPTON

The SC^{2}EPTON architecture is fully bufferless as unicast messages are sent on the SCEPTER mesh NoC, and broadcast messages are sent on the SB^2 snake ring. It supports efficient communication of coherent messages on a bufferless network, while maintaining global ordering for snoopy coherence and sequential consistency. Full-system 36-core simulations of SC^{2}EPTON on SPLASH-2 and PARSEC benchmarks show an application runtime reduction of 9%, in comparison to a baseline SCORPIO NoC, while realizing an area savings of 39% and power savings of 36%; and SMART achieves only 20% runtime reduction with significant area and power cost.

7.2 Future Work

This dissertation addresses the three challenges of scaling coherence by eliminating unscalable storage concerns, extending high performing snoopy coherence to scalable buffered mesh and bufferless ring networks, while reducing area and power substantially. As problems are tackled however, new ones emerge that can be researched further. Two potential avenues for future work, among many, are scalable buffered and bufferless coherent networks, and intelligent networks for workload adaptive behavior.

7.2.1 Scalable Buffered and Bufferless Coherent Networks

Scaling the SCORPIO network up to 100 cores reveals the need for increased buffering within the network routers to provide the necessary bandwidth for broadcasting. To ensure scalable snoopy coherent communication on these buffered networks, either broadcasts must be filtered to reduce unnecessary snoops or the network must be capable of supporting the required bandwidth. In-network coherence filtering (INCF) [11] proposed to filter redundant snoop requests by embedding small coherence filters at various routers in the network, and thereby saving network bandwidth and power. However, adapting it to the SCORPIO network is a challenge, as preliminary analysis does not reveal promising results. Transitioning from high bandwidth buffered networks to low bandwidth bufferless networks, shifts the challenge of communicating coherent broadcast requests to scheduling the network access to reduce/eliminate contention. When scaling the SB^2 snake ring to higher core counts, the fundamental scaling problem of rings is encountered. For up to 64 cores, SC^2EPTON showed performance improvement over the baseline SCORPIO, but scaling broadcasts for larger networks can be potentially achieved through the use of filtering mechanisms, optical interconnects, hierarchical networks, or new device technologies.

7.2.2 Intelligent On-Chip Networks

The SCORPIO 36-core chip prototype parameters are a result of the design space exploration and optimization using architectural simulation results. However, this is not sufficient as systems become more complex, heterogeneous, and environment or workload dependent. According to IBM's Senior Vice President, Paul Horn, the problem that can prevent the progression to the next era of computing is complexity. [6] He claims that keeping pace with Moore's law is not the chief problem, but rather managing the computing machines that have been developed thus far. Extracting the full performance potential for a varied set of applications requires runtime adaptive support as performance variation due to substrate, process, power and workload cannot be predicted and accounted for at design time. For instance, in SCEPTER, the congestion could be controlled with simple heuristics tuned prior to runtime, but required the self-throttling mechanism to obtain network state feedback and adjust the rate accordingly. Clearly the network state varies based on workload and dynamic interactions at each router, made even more unpredictable due to deflection based routing. The ideal way to optimize general-purpose chip performance would be at runtime, requiring the autonomic capability to achieve the performance self-tuning with consideration of energy monitoring feedback. State of the art multicore processors with a network interconnect and autonomic capability will be instrumental in ensuring it runs at its full potential.

BIBLIOGRAPHY

[1] ARM AMBA. http://www.arm.com/products/system-ip/amba.

[2] Open core protocol. .

[3] Oracle's SPARC T5-2, SPARC T5-4, SPARC T5-8, and SPARC T5-1B Server Architecture. http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o13-024-sparc-t5-architecture-1920540.pdf.

[4] SPLASH-2. http://www-flash.stanford.edu/apps/SPLASH.

[5] Wind River Simics. http://www.windriver.com/products/simics.

[6] Ibm autonomic computing. http://www.research.ibm.com/autonomic/, 2009.

[7] P. Abad, V. Puente, and J. Gregorio. Mrr: Enabling fully adaptive multicast routing for cmp interconnection networks. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 355–366, Feb 2009.

[8] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. Technical report, Western Research Laboratory, 1988.

[9] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Computer Architecture News*, May 1988.

[10] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator. In *ISPASS*, 2009.

[11] Niket Agarwal, Li-Shiuan Peh, and Niraj K. Jha. In-network coherence filtering: Snoopy coherence without broadcasts. In *MICRO*, 2009.

[12] Niket Agarwal, Li-Shiuan Peh, and Niraj K. Jha. In-Network Snoop Ordering (INSO): Snoopy Coherence on Unordered Interconnects. In *HPCA*, 2009.

[13] L.A. Barroso and Michel Dubois. The performance of cache-coherent ring-based multiprocessors. In *Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on*, pages 268–277, May 1993.

[14] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications, 2008.

[15] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[16] Jason F. Cantin et al. Improving multiprocessor performance with coarse-grain coherence tracking. In *International Symposium on Computer Architecture (ISCA)*, May 2005.

[17] David Chaiken, John Kubiatowicz, and Anant Agarwal. Limitless directories: A scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, 1991.

[18] K.K. Chang, R. Ausavarungnirun, C. Fallin, and O. Mutlu. Hat: Heterogeneous adaptive throttling for on-chip networks. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, 2012.

[19] Chia-Hsin Owen Chen, Sunghyun Park, Tushar Krishna, Suvinay Subramanian, Anantha P. Chandrakasan, and Li-Shiuan Peh. Smart: A single-cycle reconfigurable noc for soc applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 338–343, March 2013.

[20] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Sneger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The IBM Blue Gene/Q Interconnection Fabric. *IEEE Micro*, 32(1):32–43, 2012.

[21] Thomas Chen, Ram Raghavan, Jason N. Dale, and Eiji Iwata. Cell Broadband Engine architecture and its first implementation. In *IBM Developer Works*, November 2005.

[22] T. Chich, J. Cohen, and P. Fraigniaud. Unslotted deflection routing: a practical and efficient protocol for multihop optical networks. *Networking, IEEE/ACM Transactions on*, 2001.

[23] Pat Conway and Bill Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27:10–21, 2007.

[24] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[25] William J. Dally and Brian Towles. Route Packets, not Wires: On-Chip Interconnection Networks. *Proceedings of the Design Automation Conference (DAC)*, pages 684–689, 2001.

[26] William J. Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[27] M. Daneshtalab, M. Ebrahimi, S. Mohammadi, and A. Afzali-Kusha. Low-distance path-based multicast routing algorithm for network-on-chips. *Computers Digital Techniques, IET*, pages 430–442, 2009.

[28] Bhavya K. Daya, Chia-Hsin Owen Chen, Suvinay Subramanian, Woo-Cheol Kwon, Sunghyun Park, Tushar Krishna, Jim Holt, Anantha P. Chandrakasan, and Li-Shiuan Peh. Scorpio: A 36-core research chip demonstrating snoopy coherence on a scalable mesh noc with in-network ordering. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, 2014.

[29] R.H. Dennard, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, pages 256–268, Oct 1974.

[30] C. Fallin, C. Craik, and O. Mutlu. Chipper: A low-complexity bufferless deflection router. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011.

[31] C. Fallin, G. Nazario, Xiangyao Yu, K. Chang, R. Ausavarungnirun, and O. Mutlu. Minbd: Minimally-buffered deflection routing for energy-efficient interconnect. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 1–10, 2012.

[32] Chaochao Feng, Zhonghai Lu, Axel Jantsch, Minxuan Zhang, and Xianju Yang. Support efficient and fault-tolerant multicast in bufferless network-on-chip. *IEICE Transactions*, pages 1052–1061, 2012.

[33] Mike Galles and Eric Williams. Performance optimizations, implementation and verification of the SGI challenge multiprocessor. In *ICSC*, January 1994.

[34] Christopher J. Glass and Lionel M. Ni. The turn model for adaptive routing. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 278–287, 1992.

[35] Crispín Gómez, María E. Gómez, Pedro López, and José Duato. Reducing packet dropping in a bufferless noc. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, 2008.

[36] Paul Gratz, Chang Kim, Karthikeyan Sankaralingam, Heather Hanson, Prekishore Shivakumar, Stephen W. Keckler, and Doug Burger. On-chip interconnection networks of the trips chip. *IEEE Micro*, 27(5):41–50, 2007.

[37] A.G. Greenberg and B. Hajek. Deflection routing in hypercube networks. *Communications, IEEE Transactions on*, June 1992.

[38] B. Grot, J. Hestness, S.W. Keckler, and O. Mutlu. A qos-enabled on-die interconnect fabric for kilo-node chips. *Micro, IEEE*, 2012.

[39] F. Guderian, E. Fischer, M. Winter, and G. Fettweis. Fair rate packet arbitration in network-on-chip. In *SOC Conference (SOCC), 2011 IEEE International*, Sept 2011.

[40] M. Hayenga, N.E. Jerger, and M. Lipasti. Scarab: A single cycle adaptive routing and bufferless network. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009.

[41] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2007.

[42] Ron Ho et al. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[43] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. A 5-ghz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61, 2007.

[44] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borakar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *ISSCC*, 2010.

[45] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, Digital Equipment Corporation, 1984.

[46] N.E. Jerger, Li-Shiuan Peh, and M. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 229–240, June 2008.

[47] Daniel R. Johnson, Matthew R. Johnson, John H. Kelm, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. Rigel: A 1,024-core single-chip accelerator architecture. *IEEE Micro*, 31(4):30–41, 2011.

[48] Hanjoon Kim, Yonggon Kim, and John Kim. Clumsy flow control for high-throughput bufferless on-chip networks. *Computer Architecture Letters*, 2013.

[49] S. Konstantinidou and L. Snyder. Chaos router: architecture and performance. In *Computer Architecture, 1991. The 18th Annual International Symposium on*, 1991.

[50] T. Krishna, C.-H.O. Chen, Woo Cheol Kwon, and Li-Shiuan Peh. Breaking the on-chip latency barrier using smart. In *High Performance Computer Architecture, 2013 IEEE 19th International Symposium on*, 2013.

[51] T. Krishna and Li-Shiuan Peh. Single-cycle collective communication over a shared network fabric. In *Networks-on-Chip (NoCS), 2014 Eighth IEEE/ACM International Symposium on*, pages 1–8, Sept 2014.

[52] Tushar Krishna, Li-Shiuan Peh, Bradford M. Beckmann, and Steven K. Reinhardt. Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 71–82, 2011.

[53] Tushar Krishna, Jacob Postman, Christopher Edmonds, Li-Shiuan Peh, and Patrick Chiang. SWIFT: A SWing-reduced Interconnect For a Token-based Network-on-Chip in 90nm CMOS. In *ICCD*, 2010.

[54] Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob F. Van der Wijngaart. The case for message passing on many-core chips. In *Multiprocessor System-on-Chip*, pages 115–123. Springer, 2011.

[55] George Kurian, Jason E. Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C. Kimerling, and Anant Agarwal. Atac: A 1000-core cache-coherent processor with on-chip optical network. In *PACT*, 2010.

[56] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 1979.

[57] Milo M.K. Martin, Mark D. Hill, and David A. Wood. Timestamp snooping: An approach for extending smps. In *ASPLOS*, 2000.

[58] Milo M.K. Martin, Mark D. Hill, and David A. Wood. Token coherence: Decoupling performance and correctness. In *ISCA*, 2003.

[59] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *Computer Architecture News*, 2005.

[60] M.R. Marty and M.D. Hill. Coherence ordering for ring-based chip multiprocessors. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 309–320, Dec 2006.

[61] P.K. McKinley, Hong Xu, A.-H. Esfahanian, and L.M. Ni. Unicast-based multicast communication in wormhole-routed networks. *Parallel and Distributed Systems, IEEE Transactions on*, pages 1252–1265, 1994.

[62] G. Michelogiannakis, J. Balfour, and W.J. Dally. Elastic-buffer flow control for on-chip networks. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, 2009.

[63] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, 2010.

[64] A.K. Mishra, R. Das, S. Eachempati, R. Iyer, N. Vijaykrishnan, and C.R. Das. A case for dynamic frequency tuning in on-chip networks. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 292–303, Dec 2009.

[65] Thomas Moscibroda and Onur Mutlu. A case for bufferless routing in on-chip networks. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[66] Andreas Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *ISCA*, 2005.

[67] C.A. Nicopoulos, Dongkook Park, Jongman Kim, N. Vijaykrishnan, M.S. Yousif, and C.R. Das. Vichar: A dynamic virtual channel regulator for network-on-chip routers. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, 2006.

[68] Sunghyun Park, Tushar Krishna, Chia-Hsin Owen Chen, Bhavya K. Daya, Anantha P. Chandrakasan, and Li-Shiuan Peh. Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45nm SOI. In *DAC*, 2012.

[69] S. Rodrigo, J. Flich, J. Duato, and M. Hummel. Efficient unicast and multicast support for cmps. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 364–375, Nov 2008.

[70] Larry Seiler et al. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH*, August 2008.

[71] Korey Sewell. *Scaling High-Performance Interconnect Architectures to Many-Core Systems.* PhD thesis, University of Michigan.

[72] K. Strauss, Xiaowei Shen, and J. Torrellas. Flexible snooping: Adaptive forwarding and filtering of snoops in embedded-ring multiprocessors. In *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pages 327–338, 2006.

[73] Karin Strauss, Xiaowei Shen, and Josep Torrellas. Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors. In *MICRO*, 2007.

[74] T. Szymanski. An analysis of 'hot-potato' routing in a fiber optic packet switched hypercube. In *INFOCOM '90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration. Proceedings, IEEE*, 1990.

[75] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fee Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[76] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, June 2004.

[77] Mithuna Thottethodi, Alvin R. Lebeck, and Shubhendra S. Mukherjee. Self-tuned congestion control for multiprocessor networks. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, 2001.

[78] Mithuna Thottethodi, Alvin R. Lebeck, and Shubhendu S. Mukherjee. Blam: A high-performance routing algorithm for virtual cut-through networks. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, 2003.

[79] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 263–277, 1981.

[80] Lei Wang, Yuho Jin, Hyungjun Kim, and Eun Jung Kim. Recursive partitioning multicast: A bandwidth-efficient routing for networks-on-chip. In *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '09, pages 64–73, 2009.

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

[82] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-cpu, gpu and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264–266, Feb 2011.