Ordered Mesh Network Interconnect (OMNI): Design and Implementation of In-Network Coherence

by

Suvinay Subramanian

Bachelor of Technology in Electrical Engineering
Indian Institute of Technology Madras, 2011

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

Master of Science
in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

June 2013

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2013

Certified by .....
Li-Shiuan Peh
Associate Professor
Thesis Supervisor

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

Ordered Mesh Network Interconnect (OMNI): Design and Implementation of In-Network Coherence

by

Suvinay Subramanian

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

CMOS technology scaling has enabled increasing transistor density on chip. At the same time, multi-core processors that provide increased performance, *vis-à-vis* power efficiency, have become prevalent in a power constrained environment. The shared memory model is a predominant paradigm in such systems, easing programmability and increasing portability. However with memory being shared by an increasing number of cores, a scalable coherence mechanism is imperative for these systems. Snoopy coherence has been a favored coherence scheme owing to its high performance and simplicity. However there are few viable proposals to extend snoopy coherence to unordered interconnects - specifically, modular packet-switched interconnects that have emerged as a scalable solution to the communication challenges in the CMP era.

This thesis proposes a distributed in-network global ordering scheme that enables snoopy coherence on unordered interconnects. The proposed scheme is realized on a two-dimensional mesh interconnection network, referred to as OMNI (Ordered Mesh Network Interconnect). OMNI is an enabling solution for the SCORPIO processor prototype developed at MIT – a 36-core chip multi-processor supporting snoopy coherence, and fabricated in a commercial 45nm technology.

OMNI is shown to be effective, reducing runtime by 36% in comparison to directory and Hammer coherence protocol implementations. The OMNI network achieves an operating frequency of 833 MHz post-layout, occupies 10% of the chip area, and consumes less than 100mW of power.

Thesis Supervisor: Li-Shiuan Peh
Title: Associate Professor

Acknowledgments

The past two years at MIT have been a great experience thanks to an incredible bunch of people. At the forefront is Prof. Li-Shiuan Peh, who has been a phenomenal advisor. Her technical expertise is immense, and she engages with her students regularly, discussing interesting problems and helping us find our way in our initial years. I would sincerely like to thank her for her guidance during this project, and for her patience with me as I waded through many weeks of coding and debugging before seeing the light of the first positive results. Her unbounded enthusiasm and constantly cheerful demeanor have kept me buoyed over the last couple of years.

I am deeply greatful to my lab mates Tushar Krishna and Owen Chen, who have been great friends and wonderful mentors. As a fresh graduate student Tushar showed me the ropes of on-chip networks and computer architecture in general. In addition to his vast knowledge and experience, his constant optimism and jovial attitude have made for memorable experiences both within the lab and outside. Owen has been an amazing teacher, be it discussing concepts, reviewing code or bouncing-off research ideas. His range of knowledge – spanning low-level circuits to high-level software – and attention to detail never ceases to amaze me. He is extremely approachable, responding to my questions at any time of the day or night. He has also been a great partner through many long nights in the lab.

My lab mates Bhavya, Jason, Pablo, Sunghyun and Woo Cheol were great folks to hang out with. Some of the key ideas in this project arose out of the many hours of discussions with them. I am particularly greatful to Woo Cheol for providing the full system results for OMNI.

Outside of the lab, Anirudh has been an amazing squash partner, and a part of many engaging conversations ranging from education to pop-culture to the big philosophical questions of life. Thanks to Ujwal for being a great room mate and a part of many culinary experiments.

My parents and my brother have been a great source of support at all times in my life. Words cannot describe how indebted I am to them, for their constant encouragement, patient hearing and valuable advice.

Contents

List of Figures

List of Tables

■ 14

1 Introduction

Moore's Law has been a fundamental driver of computing, enabling increasing transistor density on-chip over the years. For the past three decades, through device, circuit, microarchitecture, architecture and compiler advances, Moore's Law coupled with Dennard's scaling [19], has resulted in commensurate exponential performance increases. In recent years, the failure of Dennard scaling, and the diminishing returns of increasing the complexity of a uniprocessor, has prompted a shift towards multi-core processors or chip multi-processors (CMPs). Figure 1-1 shows the progression from uniprocessors to multiprocessor designs over the years. This trend is expected to continue, with future processors employing tens or hundreds of cores.

1.1 On-Chip Interconnection Networks

As the number of on-chip processors increases, an efficient communication substrate is necessary to connect them.This assumes increasing importance with technology scaling for the following reasons. First, on-chip global wire delay scales upwards relative to gate delays with every technology node [27]. Thus, it takes multiple cycles for signals to travel from one edge of the chip to another, which calls for scalable and modular designs. Secondly, the increasing number of on-chip processing elements imposes the need for a high-bandwidth communication fabric. Traditional interconnects such as buses and

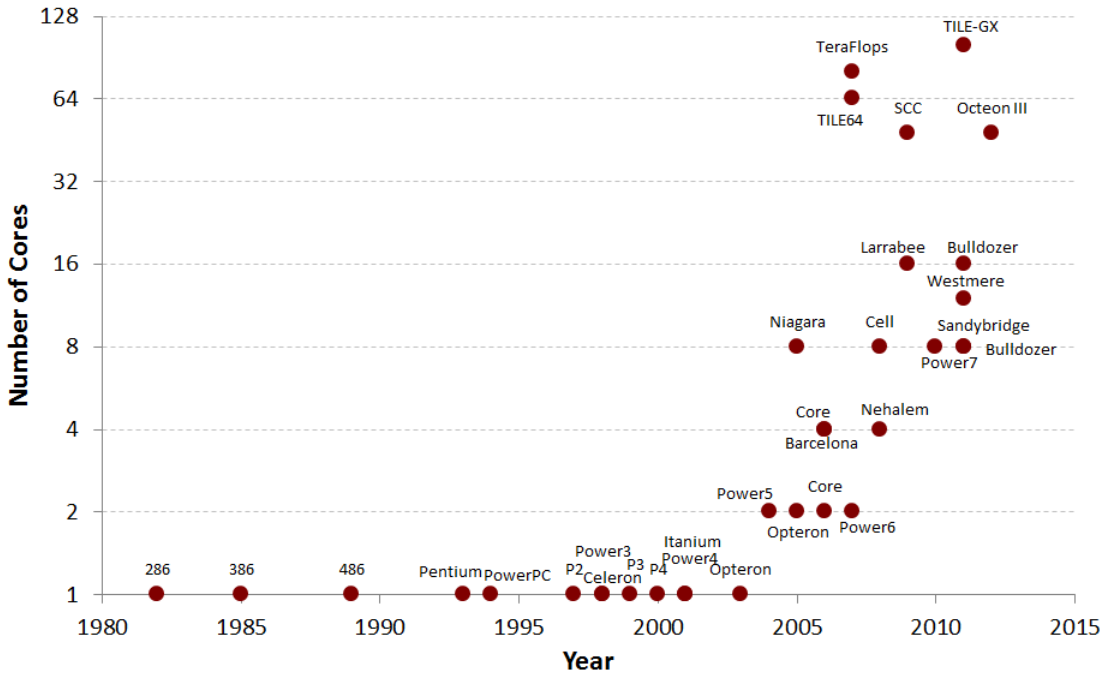


Figure 1-1: Shift to multicore

point-to-point networks are intractable for future designs - they are not modular, do not support high-bandwidth communication and make inefficient use of on-chip wires [17]. As a result, packet-switched networks-on-chip (NoC), are rapidly becoming a key enabling technology for CMPs. Such networks are composed of a topology of routers connected via point-to-point links. The routers multiplex multiple communication flows over the shared wires providing a high-bandwidth interconnect while making efficient use of the wiring resources.

On-chip interconnection networks present unique trade-offs and design constraints as compared to off-chip networks. NoCs must deliver scalable bandwidth to a large number of cores at low latency. At the same time, it is difficult to over-provision the network for performance by building large network buffers and/or using complex routing and arbitration schemes. Chip area, power consumption and design complexity are first-order design constraints in on-chip interconnection networks. In contrast to off-chip networks, where network links and transmitter-receiver circuitry are the primary latency

and power overheads, routers account for majority of the power consumption and latency overhead in on-chip interconnection networks. Therefore, a significant amount of research has focused on building low-latency and power-efficient routers.

MIT's RAW [49], UT Austin's TRIP [45], Intel's TeraFLOPs [28], Tilera's TILE64 [50], Intel Larrabee [46] and IBM Cell [15] are examples of systems that have adopted packet-switched networks.

1.2 Shared Memory in Many-Core Architectures

CMPs seek to exploit scalable thread-level parallelism (TLP) provided by applications, by using several processor cores on the same chip. The shared memory model is a predominant paradigm in such systems, easing programmability and increasing portability of applications. In a shared memory system, the processors share a single logical view of the memory, and communication between processors is abstracted as reads and writes to memory locations in the common address space. Practical realizations of shared memory multiprocessors employ a hierarchy of memories – referred to as caches – to achieve low latency access to memory, and accordingly higher application performance. These hierarchies complicate the logical view of memory since multiple copies of data could be present in different caches. A key challenge in shared memory multiprocessors is providing a "consistent" view of memory to all the processors in the presence of various memory hierarchies. Cache coherence protocols are mechanisms that address this issue, by providing a set of rules that define how and when a processor can access a memory location. The design and implementation of the cache coherence protocol is critical, to not only the correctness of the system but also its performance. With the continuing trend of increasing number of cores, a scalable coherence mechanism is imperative for shared memory systems.

The traditional approaches to cache coherence are broadcast-based snoopy protocols [26, 14, 22, 24, 29] and directory-based protocols [26, 6, 34, 36]. Snoopy proto-

cols that broadcast each coherence transaction to all cores, are attractive since they allow for efficient cache-to-cache transfers that are common in commercial workloads [10, 38]. They also do not require a directory structure which becomes expensive as the number of cores increases. However, the main limitation of these protocols is their reliance on ordered interconnects, and the bandwidth overhead of broadcasts. Directory-based protocols on the other hand rely on distributed ordering points and explicit message acknowledgements to achieve ordering. In addition they impose lower bandwidth requirement on the interconnect fabric. However, they incur a latency penalty because of directory indirection, along with the storage overhead of the directory structure on-chip.

1.3 Thesis Context and Contributions

In the many-core era, there is tension within the two classes of coherence protocols between scalability and performance. At the same time, future interconnects are likely to be modular packet-switched networks. It is desirable to have a high performance coherence mechanism, that also maps well to the underlying communication substrate.

Snoopy coherence has been a favored coherence scheme in the commercial market, for its high performance and simplicity. Hence over the years, there has been considerable effort to retain and scale snoopy protocols by adapting them to different interconnect fabrics, such as split transaction buses [22], hierarchical buses [29] and address broadcast tress [14]. However there are few viable proposals to extend snoopy protocols for unordered interconnects.

While packet switched NoCs have emerged as a promising solution to the communication challenge in many-core CMPs, there have been few implementations of on-chip networks. Further, many of these implementations either are stand-alone NoC implementations, or utilize simple NoCs as a proof-of-concept within the purview of a larger system. This is in contrast to the feature rich NoC proposals in academic literature. An associated issue is the effort in transitioning from bus-based systems to NoC-based sys-

tems – specifically the adaption required in current designs and component interfaces.

The SCORPIO (**S**noopy **CO**herent **R**esearch **P**rocessor with **I**nterconnect **O**rdering) project at the Massachusetts Institute of Technology (MIT) seeks to realize and fabricate a cache-coherent multicore chip with a novel NoC that supports in-network snoopy coherence. SCORPIO is a 36-core snoopy-cache coherent processor prototype implemented in a commercial 45-nm technology. The 36-cores are connected by a 6x6 mesh network-on-chip. The NoC provides an in-network ordering scheme that enables snoopy coherence. The NoC also presents industry standard AMBA interfaces [37] to the L2 cache controllers and memory controllers, allowing for easy integration with existing IPs, and a smoother transition from traditional interconnects.

This thesis details the design and implementation of the mesh on-chip interconnection network for the SCORPIO chip – referred to as OMNI (**O**rdered **M**esh **N**etwork **I**nterconnect).

- **Network enabled snoopy coherence:** We propose a novel scheme for enabling snoopy coherence on unordered interconnects, through a distributed in-network ordering mechanism.
- **Scalable realization of snoopy coherence:** We realize the above mechanism in a 6x6 mesh NoC, and present the design choices and microarchitectural details of the same. This prototype presents the first ever realization of snoopy coherence on a mesh network-on-chip.
- **Architectural ideas for reducing broadcast overhead:** We explore the efficacy and feasibility of filtering mechanisms in reducing broadcast overhead in snoopy coherent systems, and present microarchitectural details on how to adapt the same for OMNI.

SCORPIO is a large chip design and implementation project involving several students collaborating closely. Here, we list and acknowledge the key responsibilities and

contributions of each student: Core integration (Bhavya Daya and Owen Chen), Cache coherence protocol design (Bhavya and Woo Cheol Kwon), L2 cache controller implementation (Bhavya), Memory interface controller implementation (Owen), High-level idea of notification network (Woo Cheol), Network architecture (Woo Cheol, Bhavya, Owen, Tushar Krishna, Suvinay Subramanian), Network implementation (Suvinay), DDR2 and PHY integration (Sunghyun Park and Owen), Backend of entire chip (Owen), FPGA interfaces, on-chip testers and scan chains (Tushar), RTL functional simulations (Bhavya, Owen, Suvinay), Full-system GEMS simulations (Woo Cheol).

The rest of thesis is structured as follows. Chapter 2 presents background material on interconnection networks and cache coherence. It also contains a short summary of previous proposals for extending snoopy coherence to unordered interconnects, and prior NoC prototypes and systems. Chapter 3 provides an overview of the SCORPIO system, and presents the context of OMNI, including the requirements and characteristics of the on-chip interconnect. Chapter 4 describes the in-network ordering scheme and the design and implementation of OMNI. Chapter 5 explains how filtering mechanisms may be extended to OMNI. Chapter 6 presents evaluation results of OMNI. And finally, chapter 7 concludes the thesis and describes future directions of research on this topic.

2 Background

This chapter presents background material on interconnection networks and cache coherence protocols. While the scope of the related work and background is large, we focus on aspects that are fundamental and related to the research presented in this thesis. Section 2.1 provides an overview of on-chip interconnection networks and section 2.2 provides an overview of memory consistency and coherence. Section 2.3.1 presents a review of prior attempts at mapping snoopy coherence on unordered interconnects, and discusses their shortcomings and impediments to a realizable implementation. Finally section 2.3.2 presents an overview of real systems with NoC implementations, as well as stand-alone NoC implementations.

2.1 Interconnection Networks

An interconnection network, in broad terms, is a programmable system that transports data between different terminals. Interconnection networks occur at many scales – from on-chip networks that connect memory arrays, registers and ALUs in a processor, to system-area-networks that connect multiple processor-boards or workstations, to wide-area networks that connect multiple networks on a global scale. The focus of this work is on on-chip interconnection networks - specifically networks that serve as the communication backbone for chip-multiprocessors (CMPs).

Packet-switched networks-on-chip (NoC) are the predominant choice of interconnection networks for CMPs as they provide scalable bandwidth at relatively low latency. In general, a NoC comprises a collection of *routers* and *links* that connect various processor nodes. Routers are the components that orchestrate the communication by multiplexing the traffic flowing in from different input links onto the output links. A *message* transmission in a NoC occurs by breaking or packetizing the message into *packets* before injection into the network. A packet comprises one or more flow-control-units or *flits* which are the smallest units of the message on which flow-control is performed. A flit, in turn, is composed of one or more physical-digits or *phits* which is the number of bits that can be transmitted over a physical link in a single cycle. A message is de-packetized or re-assembled after all packets of the message have been ejected from the network. The primary features that characterize a NoC are its topology, the routing algorithm used, flow control mechanism and the router microarchitecture.

2.1.1 Topology

Network topology is the physical layout and connection of the nodes and links in the network. The topology has a significant impact on the performance and cost of the network. It determines the number of router hops a network message takes to reach its destination, as well as the wire length of each hop, thus influencing the latency of the network significantly. The number of router hops is typically captured by the *average hop count* metric. The topology also affects the *path diversity* available in the network, which refers to the number of unique paths that exist between a source and destination. A higher path diversity leads to greater robustness from failures, while also providing greater opportunities to balance traffic across multiple paths. The bisection bandwidth is an important metric determined by the topology – it captures the maximum traffic that the network can support. Figure 2-1 illustrates a few popular interconnection network topologies. The ring interconnect is simple to implement, requiring few wires and low-

radix routers. The torus is more complex to implement, but has a smaller average hop count and provides greater path diversity. The flattened butterfly requires high radix routers and long wires, but has a small average hop count and provides higher bandwidth as well.

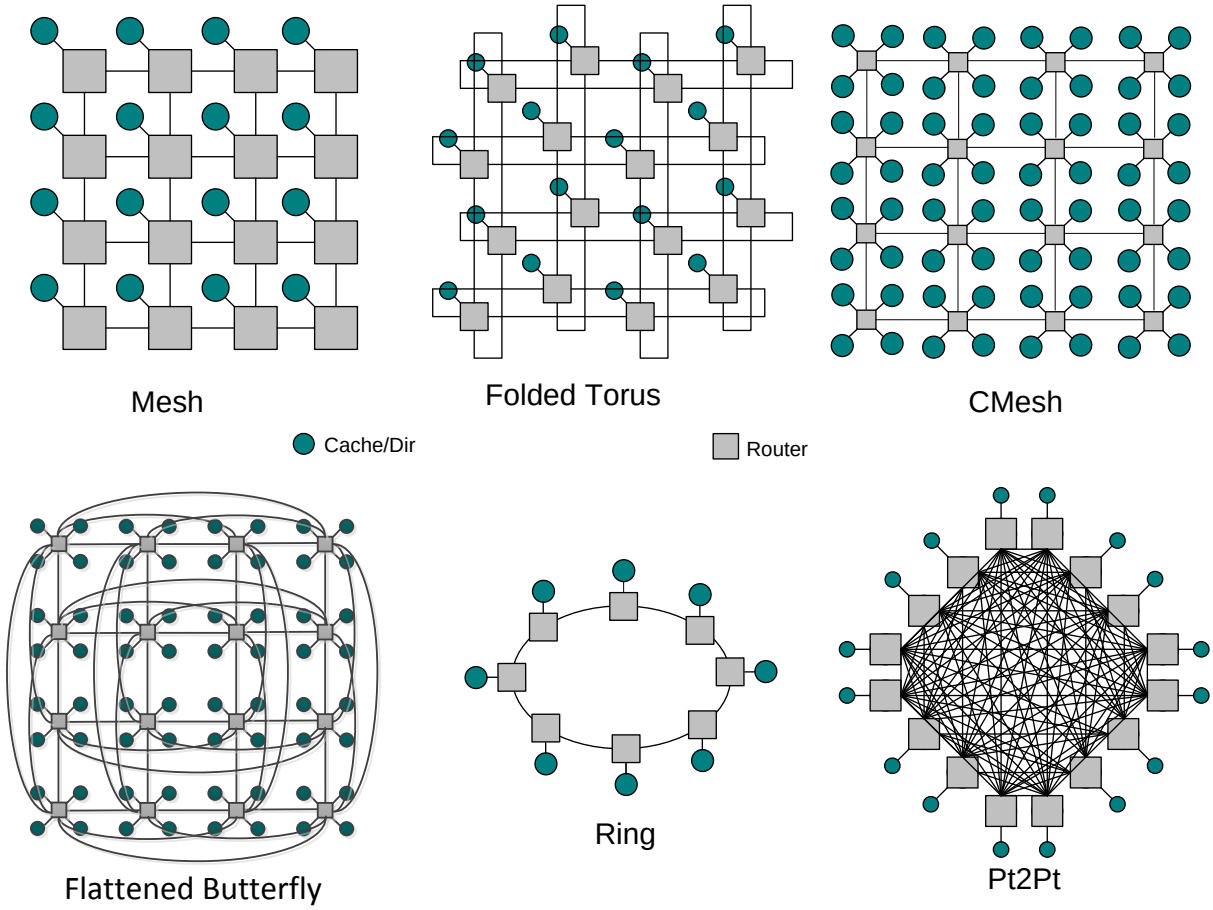


Figure 2-1: Network Topologies

2.1.2 Routing

For a given network topology, the routing algorithm determines the path a message follows through the network from the source to the destination. A good routing algorithm tries to balance network traffic evenly through the network, while minimizing contention and thereby achieving good latency. Routing algorithms may be classified

into three categories, *viz.* deterministic, oblivious and adaptive. In deterministic routing schemes, a packet always follows the same path for a given source-destination pair. They are easy to implement, and incur low hardware delay and power. In both oblivious and adaptive routing schemes, packets may traverse different paths for the same source-destination pair, the difference being that in an oblivious scheme the path is selected without regard to network state, whereas in adaptive routing schemes, the path is selected on the basis of network congestion. Another basis for classification of routing algorithms is whether they are minimal or non-minimal. Minimal routing algorithms select paths that require the fewest hops from the source to the destination, while non-minimal routing algorithms allow misroutes where packets can move away from their destination temporarily.

An on-chip routing algorithm is selected with consideration to its effect on delay, energy, throughput, reliability and complexity. In addition, the routing algorithm may also generally used to guarantee deadlock freedom. A deadlock in a network occurs when a cycle exists among the paths of multiple messages, preventing forward progress. Deadlock freedom can be guaranteed in the routing algorithm by preventing cycles among the routes generated for packets[1]. Due to the tight design constraints, on-chip networks typically use deterministic, minimal routing schemes. Dimension-ordered routing is a popular routing algorithm in this category, that also guarantees deadlock freedom. In this routing scheme messages are first routed along the X (Y) dimension and then along the Y (X) dimension. Figure 2-2 illustrates the allowed turns in DOR X-Y routing.

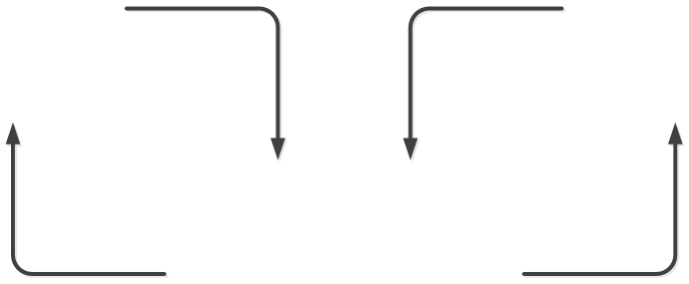


Figure 2-2: Dimension Ordered XY Routing

[1]alternatively flow control mechanisms may be used to guarantee deadlock freedom

2.1.3 Flow Control

Flow control determines how network resources such as buffers and channels are allocated to packets. A good flow control mechanism allocates these resources in an efficient manner so the network achieves high throughput and good latency. Flow control mechanisms are classified by the granularity at which resource allocation occurs.

Circuit-switching pre-allocates links across multiple hops for the entire message. Prior to sending the message, a probe is sent out which reserves all the links from the source to the destination. Circuit-switching has the advantage of low latency and being bufferless. However it leads to poor bandwidth utilization. Packet based flow control breaks down messages into packets and interleaves them on links, thereby improving utilization. Store-and-forward and virtual-cut-through are packet based flow control schemes.

Wormhole and virtual channel flow control are examples of flit-level flow control schemes. Packets are broken down into flits, and buffers and channel bandwidth are allocated at the flit granularity. In wormhole flow control, flits are allowed to move to the next router before the entire packet is received at the current router. This results in efficient buffer utilization. However, it is susceptible to head-of-the-line blocking, wherein a flit at the head of a buffer queue is unable to secure its desired channel, and thereby prevents flits behind it from making progress. Virtual-channel (VC) flow control also allocates resources at the flit granularity and has the benefits of wormhole flow control. It also addresses the poor bandwidth utilization problem in wormhole flow control by using virtual channels. A virtual channel is essentially a separate buffer queue in the router, and multiple VCs share the same physical links. Since there are multiple VCs for every physical channel, even if one channel is blocked, other packets can use the idle link. This prevents head of the line blocking. VCs can also be used to guarantee deadlock freedom in the network. In cache-coherent systems, VCs are often utilized to break protocol-level deadlocks.

2.1.4 Router Microarchitecture

Routers must be designed to meet the latency and throughput requirements within tight area and power budgets. The router microarchitecture impacts the per-hop latency and thus the total network latency. The microarchitecture also affects the frequency of the system and area footprint. The microarchitecture of a simple router for a two-dimensional mesh network is shown in figure 2-3. The router has five input and output ports, corresponding to its four neighboring directions north (N), south (S), east (E), west (W), and local port (L), and implements virtual-channel (VC) flow control. The major components in the router are the input buffers, route computation logic, virtual channel allocator, switch allocator and crossbar switch. A typical router is pipelined, which allows the router to be operated at a higher frequency. In the first stage, the incoming flit is written into the input buffers (BW). In the next stage, the routing logic performs route compute (RC) to determine the output ports for the flit. The flit then arbitrates for a virtual channel (VC) corresponding to its output port. Upon successful arbitration for a VC, it proceeds to the switch allocation (SA) stage where it arbitrates for access to the crossbar. On winning the required output port, the flit traverses the crossbar (ST), and finally the link to the next node (LT).

2.2 Memory Consistency and Cache Coherence

With the emergence of many-core CMPs, the focus has shifted from single threaded programs to parallel programs. The shared memory abstraction provides several advantages, by presenting a more natural transition from uniprocessors, and by simplifying difficult programming tasks such as data partitioning and load distribution. The use of multi-level caches can substantially reduce the memory bandwidth demand, and offer fast access to data. However, this introduces the problem of presenting a "correct" view of memory to all the processors, in the presence of multiple copies of data in different processor caches. Shared memory correctness is generally divided into two sub-issues, *viz.* consistency

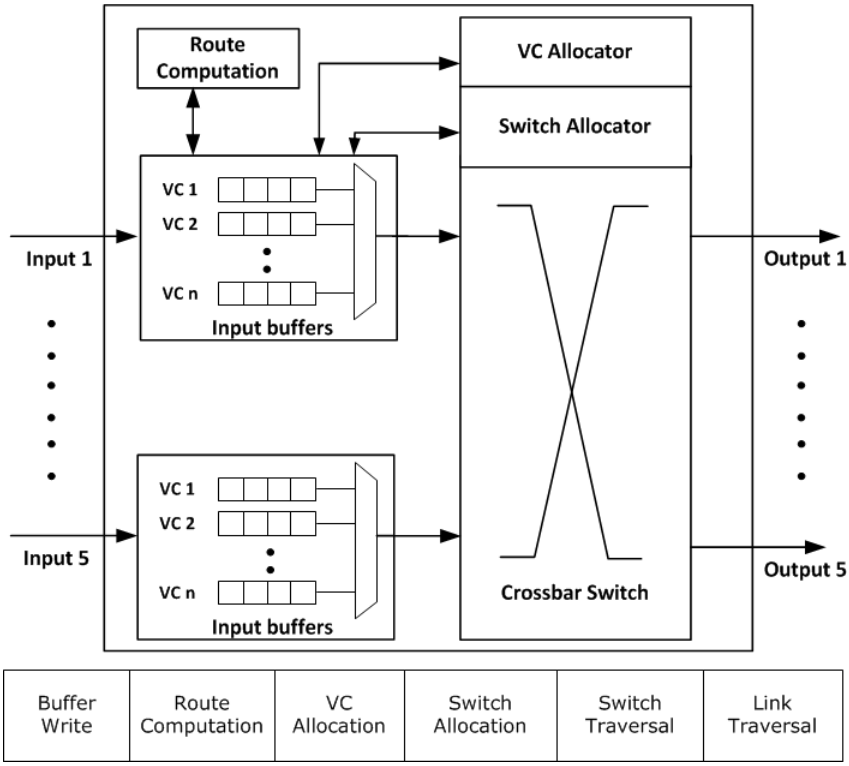


Figure 2-3: Microarchitecture and pipeline of router

and coherence. Broadly, coherence is concerned with reads and writes to a single memory location. It determines what value can be returned by a read. Consistency on the other hand is concerned with ordering of reads and writes to multiple memory locations. While coherence is not a necessity for correct shared memory implementations, in most real systems, coherence is used as an enabling mechanism for providing the appropriate memory consistency. We provide an overview of these two issues within the context of the subject matter of this thesis.

2.2.1 Memory Consistency Overview

Single threaded serial programs present a simple and intuitive model to the programmer. All instructions appear to execute in a serial order[2], and a thread transforms a given input state into a single well-defined output state. The consistency model for a single threaded

[2]in an out-of-order core the hardware might execute instructions in a different order

program on a uniprocessor, therefore, maintains the invariant, that a load returns the value of the last store to the corresponding memory location.

Shared memory consistency models are concerned with the loads and stores of multiple threads running on multiple processors. Unlike uniprocessors, where a load is expected to return the value from the last store, in multiprocessors the most recent store may have occurred on a different processor core. With the prevalence of out-of-order cores, and multiple performance optimizations such as write buffers, prefetching etc., reads and writes by different processors may be ordered in a multitude of ways. To write correct and efficient shared memory programs, programmers need a precise notion of how memory behaves with respect to reads and writes from multiple processors. Memory consistency models define this behavior, by specifying how a processor core can observe memory references made from other processors in the system. Unlike the uniprocessor single threaded consistency model which specifies a single correct execution, shared memory consistency models often allow multiple correct executions, while disallowing many more incorrect executions.

While there are many memory consistency models for multiprocessor systems, arguably the most intuitive model is *sequential consistency*. Lamport [35] was the first to formalize the notion of sequential consistency. A single processor core is said to be *sequential* if the result of an execution is the same as if the operations had been executed in the order specified by the program. A multiprocessor system is said to be 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 the order specified by the program. This total order of memory operations is referred to as *memory order*. Figure 2-4 depicts a code segment involving two processors $P1$ and $P2$. The *critical section* could be a portion of code that attempts to access a shared resource that must not be accessed concurrently by more than one thread. When $P1$ attempts to enter the critical section, it sets the `flag1` to 1, and checks the value of `flag2`. If `flag2` is 0, then it implies $P2$ has not tried to enter the critical section, and

```
P1 : flag1 = 0;                P2 : flag2 = 0;
        …                              …
        …                              …
     flag1 = 1;                     flag2 = 1;
L1 : if (flag2 == 0)           L1 : if (flag1 == 0)
           critical section               critical section
```

Figure 2-4: Example for sequential consistency

therefore it is safe for $P1$ to enter. The assumption here is that if `flag2` is read to be `0`, then it means $P2$ has not written `flag2` yet, and consequently not read `flag1` either. However, since processor cores apply several optimizations for improving performance, such as out-of-order execution, this ordering may not hold true. Sequential consistency ensures this ordering by requiring that the program order among operations by $P1$ and $P2$ be maintained. Sequential consistency conforms to the typical thought process when programming sequential code, and thus presents a very intuitive model to programmers to reason about their programs.

2.2.2 Cache Coherence Protocols

In multiprocessor systems, the view of memory held by different processors is through their individual caches, which, without any additional precautions could end up seeing two different values. Figure 2-5 illustrates the coherence problem. We assume write-through caches in the example. If processor B attempts to read value X after time 3, it will receive 1 which is incorrect.

Time	Event	Cache Contents CPU A	Cache Contents CPU B	Memory contents for loc. X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A writes 0 to X	0	1	0

Figure 2-5: Cache coherence problem for a single memory location X

Informally, a memory system is said to be coherent if any read, returns the most recently read value of the data item. Formally, coherence is enforced by maintaining the following invariants in the system:

1. **Single-Writer, Multiple-Read Invariant** For any memory location A, at any given (logical) time, there exists only a single core that may write to A (and can also read it), or some number of cores that may only read A.

2. **Data-Value Invariant** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

The typical mechanism for enforcing these variants is as follows. Cache coherence protocols attach permissions to each block stored in a processor's cache. On every load or store, the access is sent to the local cache and an appropriate action is taken after looking at the permission for that block. These permissions are referred to as the state of the cache line. The commonly used stable states[3] in a cache are described below.

- *Invalid (I):* The block in the cache is not valid. The cache either does not contain the block, or it is a potentially stale copy that it cannot read or write.

- *Shared (S):* The block in the cache is valid, but the cache may only read this block.

[3]the states are often named in accordance to the characteristics that they capture, about the particular cache block

This state typically implies that other processor caches in the system may be sharing this block with read permission.

- *Modified (M):* The processor cache holds the block, and has read and write access to the block. This also implies that this cache holds the only valid copy of the block in the system, and must therefore respond to requests for this block.

In addition to the above states, two additional states may be added to improve performance – the *Exclusive (E)* state assigned to a block on a read request if there are no copies cached at other caches, allows for implicit write access, thereby saving time, and the *Owned (O)* state assigned to a block indicates that this cache will source a remote coherence request, which prevents slow DRAM access. Most cache coherence protocols allow cache blocks to transition to a different state through a set of transactions. Table 2.1 lists a set of common transactions and the associated goals of the requester.

Table 2.1: Common Coherence Transactions

Transaction		Goal of requester
GetShared	(GetS)	Obtain block in Shared (read-only) state
GetModified	(GetM)	Obtain block in Modified (read-write) state
Upgrade	(Upg)	Upgrade block from read-only to read-write
PutShared	(PutS)	Evict block in Shared state
PutExclusive	(PutE)	Evict block in Exclusive state
PutOwned	(PutO)	Evict block in Owned state
PutModified	(PutM)	Evict block in Modified state

Throughout literature, there are two major approaches to cache coherence protocols, *viz.* broadcast based snoopy protocols and directory-based protocols.

1. **Broadcast-based snoopy protocols** These protocols rely on a simple idea: all coherence controllers observe (*snoop*) all coherence requests in the same order, and collectively take the correct action to maintain coherence. By requiring that all requests to a block arrive in order, a snooping system enables the distributed coherence controllers to correctly update the finite state machines that collectively rep-

resent a cache block's state. Snoopy protocols rely on ordered interconnects, like bus or tree networks, to ensure total ordering of transactions. A total ordering of transactions subsumes the per-block orders thus maintaining coherence. Additionally, the total ordering makes it easier to implement memory consistency models that require a total ordering of memory transaction, such as sequential consistency.

2. **Directory-based protocols** The key principle in directory-based protocols is to establish a *directory* that maintains a global view of the coherence state of each block. The directory tracks which caches hold each block and in what states. A cache controller that wants to issue a transaction sends the same to the directory, and the directory looks up the state of the block and determines the appropriate action. A sharer list for each block enables the directory to avoid broadcasting messages to all nodes, instead sending targeted messages to the relevant nodes. Like snoopy protocols, directory protocols also need to define when and how coherence transactions are ordered with respect to other transactions. In most directory protocols, the transactions are ordered at the directory. However, additional mechanisms are required to enforce sequential consistency.

Directory protocols have the advantage of requiring low communication bandwidth and are thus scalable to large number of cores. However the scalability comes at the cost of higher latencies caused by directory indirection. Further, as the number of cores increases, maintaining a directory incurs significant area and power overhead. In addition, directory protocols are harder to implement and require significant verification effort to enforce memory consistency guarantees.

Snoopy protocols have traditionally dominated the multiprocessor market. They enable efficient direct cache-to-cache transfers that are common in commercial workloads [10, 38], have low overheads in comparison to directory-based protocols and are simple to design. In addition, the total ordering of requests in snoopy coherence, enables easy implementation of desirable consistency guarantees, such as sequential consistency. However, there are two primary impediments in scaling snoopy coherence to many-core

CMPs, *viz.* broadcast overhead and reliance on ordered interconnects.

As we scale to many-core CMPs, a scalable coherence mechanism that maps well to the underlying communication substrate is imperative. Both snoopy and directory protocols have advantages that one would like to see in an ideal many-core CMP cache coherence protocol. However, there are limitations in each that prohibit their direct adoption on CMPs. In this thesis, we attempt to enable scalable snoopy coherence for many-core CMPs by overcoming its shortcomings through in-network techniques. The philosophy of off-loading coordination and distributed decisions to the network is not new. It has been argued in few prior works [20, 21, 9] that leveraging the network effectively may lead to better system design. In section 2.3.1, we review previously proposed techniques that extend snoopy coherence to unordered interconnects, and discuss their shortcomings and impediments to efficient implementation.

2.3 Related Work

2.3.1 Snoopy coherence on unordered interconnects

Uncorq [48] is an embedded ring coherence protocol in which snoop requests are broadcast to all nodes, using any network path. However along with the snoop request broadcast, a response message is initiated by the requester. This response message traverses a logical ring, collecting responses from all the nodes and enforcing correct serialization of requests. However, this method requires embedding of a logical ring onto the network. In addition, there is a waiting time for the response to traverse the logical ring. Further, Uncorq only handles serialization of requests to the same cache line – which is sufficient to enforce snoopy coherence, but does not enforce sequential consistency.

Multicast snooping [12] uses totally ordered fat-tree networks to create a global ordering for requests. Delta coherence protocols [51] also implement global ordering by using isotach networks. However, these proposals are restricted to particular network topologies.

Time-stamp snooping (TS) [38] is a technique that assigns logical time-stamps to requests and re-orders packets at the end points. It defines an ordering time (OT) for every request injected into the network. Further each node maintains a guaranteed time (GT), which is defined as the logical time that is guaranteed to be less than the OTs of any requests that may be received later by a node. Once a network node has received all packets with a particular OT, it increments its GT by 1. Nodes exchange tokens to communicate when it is safe to update their GT. Since multiple packets may have the same OT, each destination employs the same ordering rule to order such packets. While TS allows snoopy coherence to be implemented on unordered interconnects, it has a few drawbacks. Each node needs to wait for all packets with a particular OT to arrive before it can process them and update its GT. This can increase the latency of packets. In addition, it also requires large number of buffers at the end-points which is not practical. TS also requires updating of slack values of messages buffered in the router, resulting in additional ports in the router buffers.

In-Network Snoop Ordering (INSO) [9] maps snoopy coherence onto any unordered interconnect, by ordering requests in a distributed manner within the network. All requests are tagged with distinct snoop orders which are assigned based on the originating node of the request. Nodes process messages in increasing sequence of snoop orders. However, INSO requires unused snoop orders to be expired periodically, through explicit messages. If the expiration period is not sufficiently small, then it can lead to degradation of performance. On the other hand, small expiration periods also leads to increased number of expiry messages, especially from nodes that do not have any active requests. This consumes power and network bandwidth, which is not desirable for practical realizations. While the INSO proposal suggests using a separate network for these expiry messages, it can lead to subtle issues where expiry messages overtake network messages, which could void the global ordering guarantee.

Intel's QPI [1] and AMD's Hammer [16] protocol are industrial protocols that extend snoopy coherence for many-core processors. AMD Hammer is similar to snoopy

coherence protocols in that it does not store any state on blocks stored in private caches. It relies on broadcasting to all tiles to solve snoop requests. To allow for implementation on unordered interconnects, the Hammer protocol introduces a home tile that serves as an ordering point for all requests. On a cache miss, a tile sends its request to the home tile which then broadcasts the same to all nodes. Tiles respond to requests by sending either an acknowledgement or the data to the requester. A requester needs to wait for responses from all tiles, after which it sends an unblock message to the home tile, thereby preventing race conditions. While this protocol allows for snoopy coherence on unordered interconnects, the ordering point indirection latency can become prohibitively large as the core count increases. Intel's QPI implements a point-to-point interconnect that supports a variant of snoopy coherence, called *source snooping*. QPI also introduces a home node for ordering coherence requests. However, they add a new state *Forward* to their coherence protocol to allow for fast transfer of shared data. QPI is well suited for a small number of nodes, and scales well for hierarchical interconnects [23].

2.3.2 NoC Prototypes and Systems

MIT's RAW [49] is a tiled multicore architecture that uses a 4x4 mesh interconnection network to transmit scalar operands. However, this network is also used to carry memory traffic between the pins and the processor for cache refills. The TRIPS [25] processor uses an on-chip operand network to transmit operands among the ALU units within a single processor core. It also uses a 4x10 wormhole routed mesh network to connect the processor cores with the L2 cache banks and I/O controllers.

IBM's Cell [15] is a commercial processor that uses ring interconnects to connect its processing elements, external I/O and DRAM. It uses four ring networks and uses separate communication paths for commands and data. The interconnect supports snoopy coherence, and uses a central address concentrator that receives and orders all broadcast requests [30].

Tilera's TILE64 [50] is a multiprocessor consisting of 64 tiles connected together by

five 2D mesh networks. The five mesh network support distinct functions and are used to route different types of traffic. Four of the five networks are dynamically routed and implement wormhole routing, while the static network is software scheduled.

Intel's Teraflops [28] research chip demonstrated the possibility of building a high speed mesh interconnection network in silicon. The prototype implements 80 simple RISC-like processors, each containing two floating point engines. The mesh interconnect operates at 5 GHz frequency, achieving performance in excess of a teraflop.

Intel's Single Chip Cloud computer (SCC) [2] connects 24 tiles each containing 2 Pentium processors through a 4x6 mesh interconnect. The interconnect uses XY routing and carries memory, I/O and message passing traffic. The NoC is clocked at 2 GHz and consumes 6W of power. The hardware is not cache-coherent, and instead supports the message passing programming model.

2.4 Chapter Summary

In this chapter we provided an overview of interconnection networks. We introduced some basic terminology, and discussed common techniques used in the design of on-chip interconnection networks. We also reviewed a few important concepts pertaining to cache coherence and memory consistency. We discussed the benefits and drawbacks of the two broad classes of coherence protocols *viz.* snoopy and broadcast protocols, and pointed towards the need for scalable coherence mechanisms in future CMP designs. We then considered a few recent approaches at tackling the scalable coherence problem and present some of the issues in their adoption. And finally, we provided an overview of NoC prototypes and systems that employ scalable on-chip interconnection networks.

3 System Overview

The shift to multicore has brought a whole host of new challenges. In the CMP era, communication has become a first order design constraint. Two key issues that future CMP designs have to contend with are, the need for a scalable coherence mechanism, and an efficient communication fabric. Snoopy coherence is valued for being high performance and simple to design, while packet-switched NoCs are rapidly becoming an enabling technology for CMP designs. Current cores, caches and memory controllers designed for snoopy coherence target traditional interconnects, such as buses and crossbars, that will not scale. There have been few viable proposals to extend snoopy coherence to packet switched interconnects. Even among such proposals, it is unclear what would be the effort of transitioning current systems to NoC based systems.

SCORPIO (**S**noopy **CO**herent **R**esearch **P**rocessor with **I**nterconnect **O**rdering) is a 36-core snoopy-cache coherent processor prototype implemented in commercial 45-nm technology. The 36-cores are connected in a 6x6 mesh NoC with a novel mechanism for supporting snoopy coherence. The NoC supports standard Advanced Microcontroller Bus Architecture interfaces, allowing easy interchange and replacement of traditional interconnects with a faster, scalable interconnect without significant redesign on other system components. This chapter provides an overview of the SCORPIO system. We focus on key components in the system, and specifically their interactions with the on-chip interconnect. The scope and details of the entire system is much larger; we restrict ourselves

to factors that impact the network design, attempting to provide a context for the various design decisions, various requirements, and various enabling features of the on-chip interconnection network of SCORPIO.

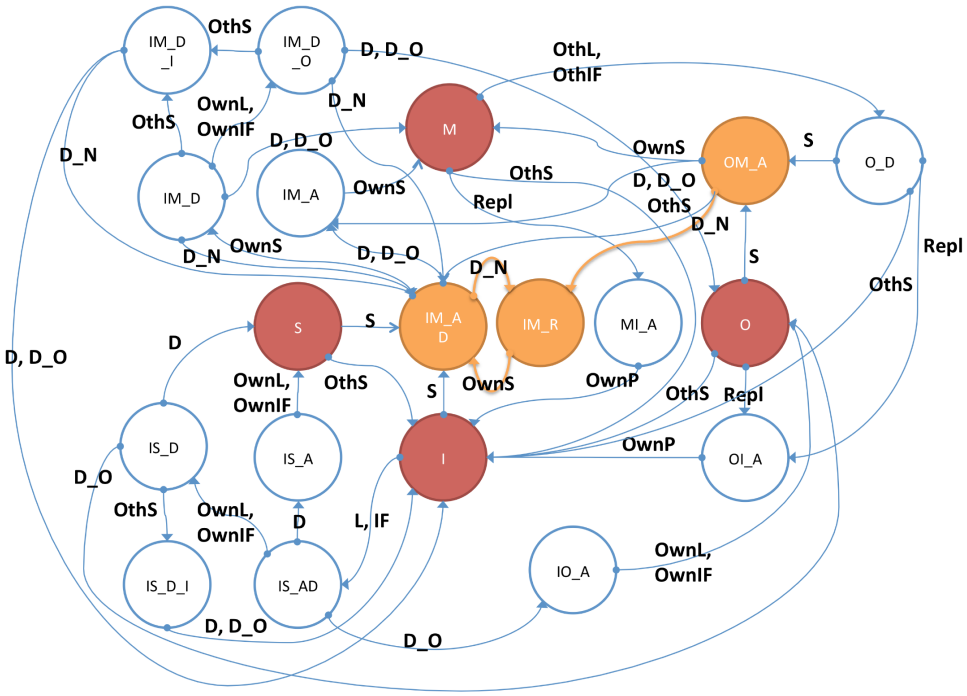
3.1 Key Components

The SCORPIO chip contains 36 tiles connected through a 6x6 mesh network. Each tile includes a Freescale e200z760 processor, split L1 I/D caches, a private L2 cache supporting snoopy coherence protocol, a network interface and router that support in-network global ordering. The chip includes two dual-ported memory controllers, each of which connects to a Cadence DDR PHY module and off-chip DIMM modules. The 6x6 mesh NoC used to connect the 36 tiles implements a novel scheme for providing a global ordering of messages, and enabling snoopy coherence. Standard Advanced Microcontroller Bus Architecture (AMBA) interfaces are employed between the network, L2 cache, processor core and memory controllers, decomposing the architecture into independent modules and allowing for easy integration of components.

Cores: The processor core has a split instruction and data 16KB L1 cache with independent AMBA AHB ports. The AHB protocol supports a single read or write transaction at a time. Transactions between pending requests from the same AHB port are not permitted, restricting the number of outstanding misses per core to two (one data cache miss and one instruction cache miss).

L2 Cache and Snoopy Coherence: Each tile contains a 128 KB private L2 cache that is maintained coherent through a modified MOSI snoopy coherence protocol. Figure 3-1 shows the state transition diagram for the coherence protocol.

Barriers and Synchronization: While the interconnection network of the SCORPIO



L	Load
S	Store
IF	IFetch
Repl	L2 Replacement
OwnL	Own network Load
OwnS	Own network Store
OwnIF	Own network IFetch
OwnP	Own network Writeback
OthL	Other network Load
OthS	Other network Store
OthIF	Other network IFetch
OthP	Other network Writeback
D	Data Response
D_O	Data Response – Owner
D_N	Data Response - Null

Figure 3-1: Cache coherence protocol for SCORPIO

processor supports global ordering, the cores themselves support only a weak consistency model. Hence, the processor cores utilize an `msync` instruction for barrier synchronization. These instructions need to be seen in global order by all cores in the system. Upon receiving an `msync` request, the cores take appropriate action and respond with an ACK to the original requester.

Memory Interface Controller: The memory interface controller is responsible for interfacing with the memory controller IP from Cadence. It maintains one bit per cache line contained in a directory cache, indicating if the memory is the owner of that cache line or not. As it snoops requests in the network, it uses this information to determine if it is required to respond or not.

3.2 OMNI: Ordered Mesh Network Interconnect

The on-chip interconnection network used in the SCORPIO processor is referred to as the Ordered Mesh Network Interconnect (OMNI). OMNI is a key enabler for SCORPIO's coherence and consistency guarantees. OMNI provides for global ordering of coherence requests, which also enforces sequential consistency in the network. In addition, it supports different message classes and implements multiple virtual networks to avoid protocol-level deadlocks. We describe the different virtual networks supported by OMNI, and the characteristics of these virtual networks below.

1. **Globally Ordered Request (GO-REQ):** Messages on this virtual network are globally ordered and broadcast to all nodes in the system. Coherence requests travel on this virtual network, as do `msync` requests.

2. **Point-to-point Ordered Request (P2P-REQ):** This virtual network supports point-to-point ordering of messages. Non-coherent requests to the memory controllers are handled on this virtual network.

3. **Unordered Response (UO-RESP):** This virtual network supports unicast packets, and messages are unordered. Coherence responses and `msync` ACKs are sent on this virtual network. The separation of the coherence and `msync` responses from the corresponding requests prevents protocol-level deadlock.

Figure 3-2 is a schematic of the SCORPIO system with OMNI highlighted.

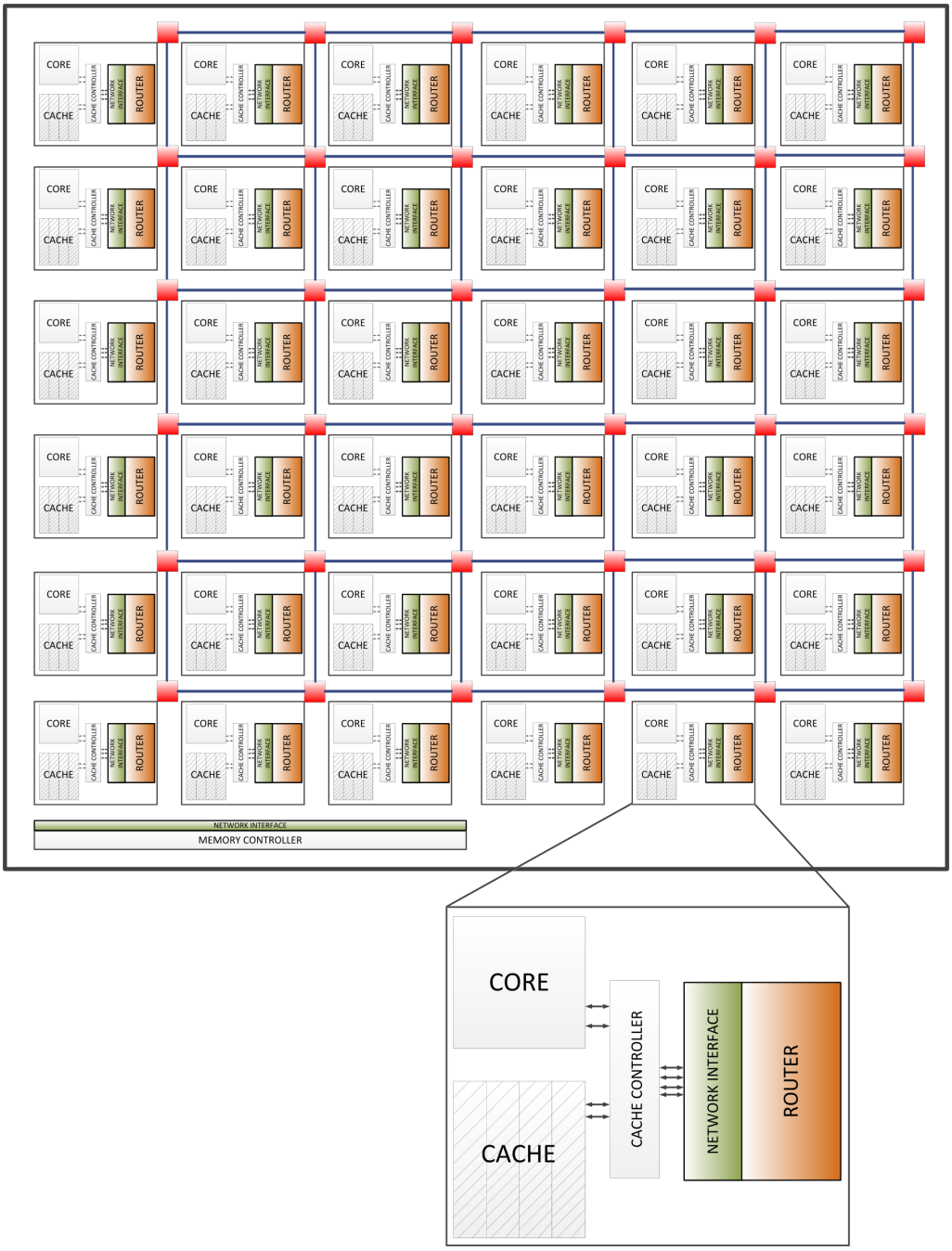


Figure 3-2: SCORPIO processor schematic with OMNI highlighted

4 OMNI: Design and Implementation

This chapter describes the design and implementation of the Ordered Mesh Network Interconnect (OMNI) of the SCORPIO processor. Section 4.1 presents an overview of the in-network global ordering scheme, highlighting some of the key characteristics of the scheme. Section 4.2 describes how the above scheme is implemented in the network, delving into the microarchitectural details of the router and network interface controller.

4.1 OMNI: Overview

It has been shown previously that snoopy protocols depend on the logical order and not the physical time at which requests are processed [12, 38], i.e. the physical time at which requests are received at the nodes are unimportant so long as the global order in which all nodes observe requests remains the same. Traditionally, global ordering on interconnects have relied on a centralized arbiter or a global ordering point. In such a scheme, all messages are first sent to the ordering point, which then broadcasts the messages to the rest of the system. The interconnection network needs to ensure that the order in which messages leave the ordering point is the same as the order in which the nodes observe the requests. However, as the number of nodes increases, such a mechanism introduces greater indirection latency. Forwarding to a single ordering point also creates congestion at the ordering point degrading network performance.

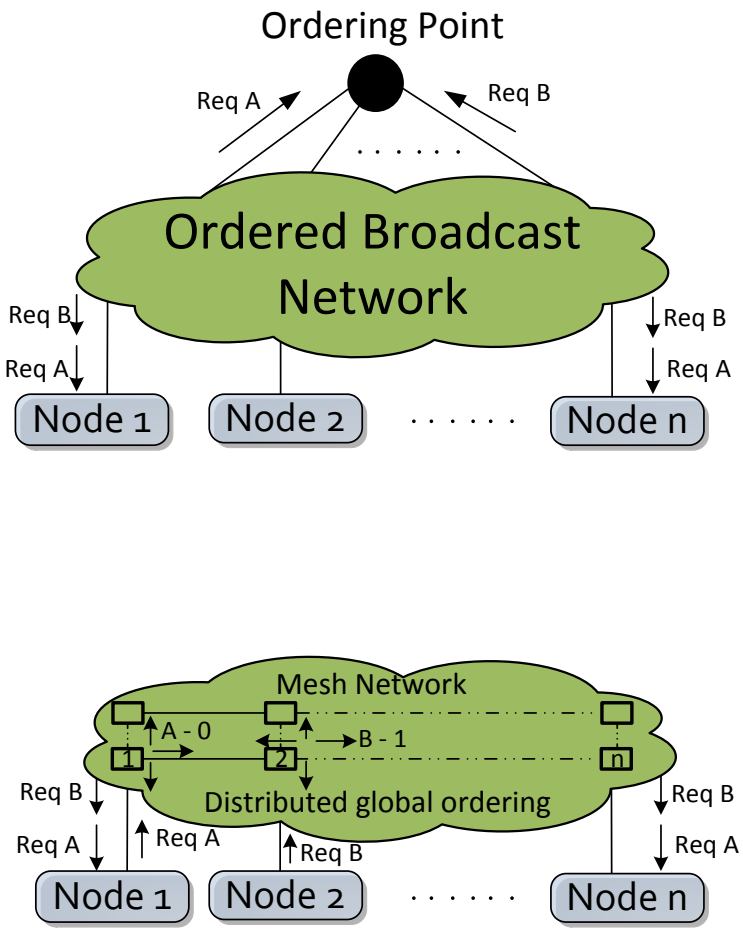


Figure 4-1: Ordering Point vs OMNI

In OMNI, we remove the requirement of a central ordering point, and instead, entrust this responsibility to each individual node. Fundamentally, a decision on ordering essentially involves deciding which node to service next. We allow all nodes in the system to take this decision locally, but guarantee that the decision is consistent across all the nodes. At synchronized time intervals - referred to as *time window*, all nodes in the system perform a local decision on *which nodes to service next*, and *in what order to service these nodes*. The mechanism for the same is as follows. Consider a system with N nodes numbered from 1 through N – referred to as *source ID (SID)*. The nodes are connected by an interconnection network of a particular topology; we refer to this network as the *main network*. All messages from a particular node are tagged with the SID. Now, for every

message injected into the main network we construct a corresponding notification message. This message essentially contains the SID, and indicates to any node that receives this notification message, that it should expect a message from the corresponding source. The notification message is sent to all nodes through a separate "fast" network (we clarify what "fast" entails later) – we refer to this "fast" network as the *notification network*. At the end of each synchronized time interval, we guarantee that all nodes have received the same set of notification messages. For every node, this establishes the set of sources they will service next. Every node performs a local decision on the order in which it will service these sources – for example, the decision rule could be "increasing order of source ID". Consequently, this fixes the global order for the actual messages in the system. This global order is captured through a counter maintained at each node called the *expected source ID* (ESID). Messages travel through the *main network* and are delivered to all nodes in the system. However, they are processed by the network interface (NIC) at every node in accordance to the global order.

Table 4.1: Terminology

Term	Meaning
N	Number of nodes in the system. For SCORPIO $N = 36$
Main Network	6x6 mesh network supporting 3 virtual networks All network messages are carried on this network
Notification Network	6x6 lightweight, contention-free mesh network Carries notification messages, used for global ordering
Time Window	Synchronized time intervals at which nodes in the system take a decision on ordering of requests
Source ID (SID)	Source of a message – typically the node number
Expected SID (ESID)	Counter capturing the global order, indicates the source of the next message to be processed
Processing a message	Dequeuing a received packet in accordance to global order, followed by parsing of the packet

4.1.1 Walkthrough example

We present a detailed walkthrough of OMNI here. For simplicity, the walkthrough example considers a 16-tile CMP system. Each tile comprises a processor core with a private L1 cache, and a private L2 cache attached to a router. Two memory controllers on two sides of the chip are connected to the routers as well. Cache coherence is maintained between the L2 caches and the main memory. The routers are connected in a 4x4 packet-switched mesh network, i.e. the *main network* is a 4x4 mesh. The "fast" notification network, in this example, is a *contention-free* light-weight 4x4 mesh network (the details of the implementation are presented later) We walkthrough how two requests $M1$ and $M2$ are handled and ordered by OMNI.

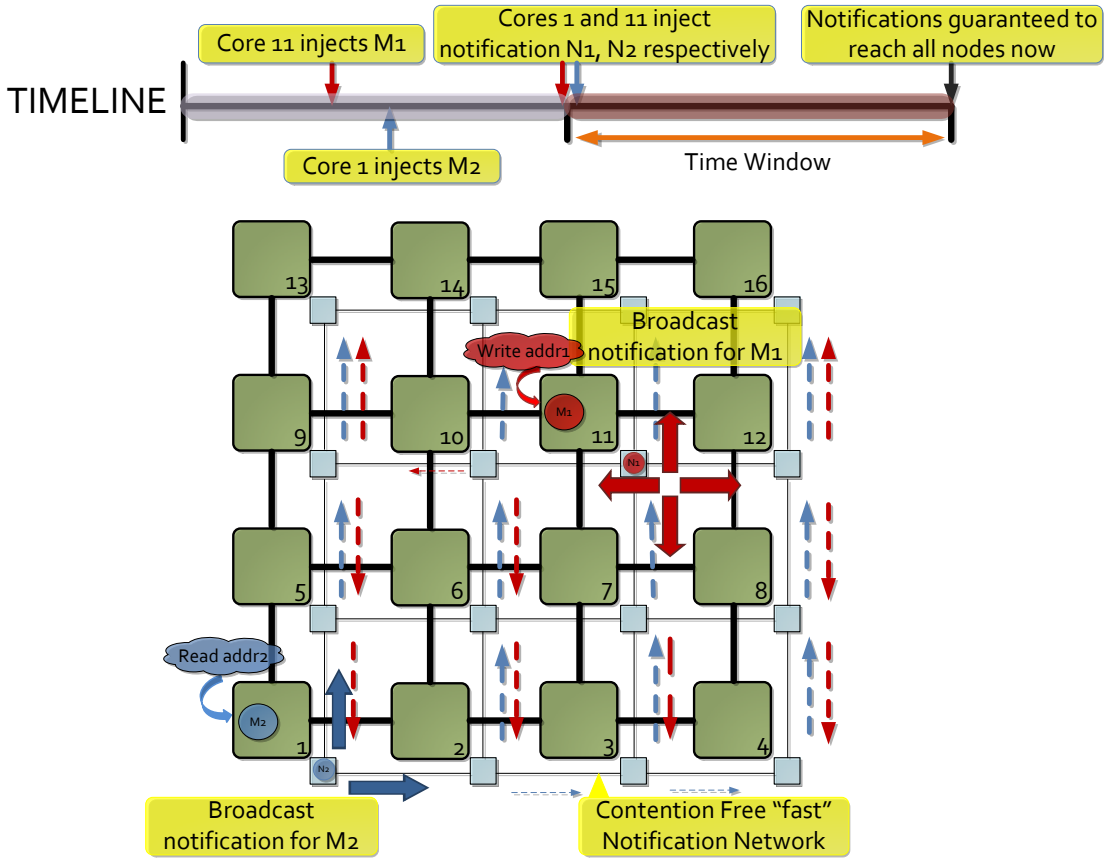


Figure 4-2: Cores 11 and 1 inject $M1$ and $M2$ immediately into the main network. The corresponding notifications $N1$ and $N2$ are injected into the notification network at the start of the next time window

1. Core 11's L2 cache miss triggers a request `Write addr1` to be sent to its cache controller, which leads to message $M1$ being injected into the network through the NIC. The NIC takes a request from the cache controller and encapsulates it into a single-flit packet and injects the packet into the attached router in the main network. Corresponding to $M1$, notification message $N1$ is generated. However notification messages are injected into the notification network only at the beginning of every time window. Therefore notification message $N1$ is sent out at the start of the next time window.

2. Similarly, Core 1's L2 cache miss triggers request `Read addr2` to be sent to its cache controller. This leads to message $M2$, being injected by the NIC into the main network. The corresponding notification message $N2$ is generated, and injected at the start of the next time window as shown in figure 4-2

3. Messages $M1$ and $M2$ make their way through the main network, and are delivered to all nodes in the network. However, until the corresponding notifications are received and processed by the nodes, these messages have not been assigned a global order. Hence, they are held in the NIC or routers of the destination nodes.

4. At the same time, notifications $N1$ and $N2$ make their way through the "fast" notification network, and are delivered to all the nodes in the network.

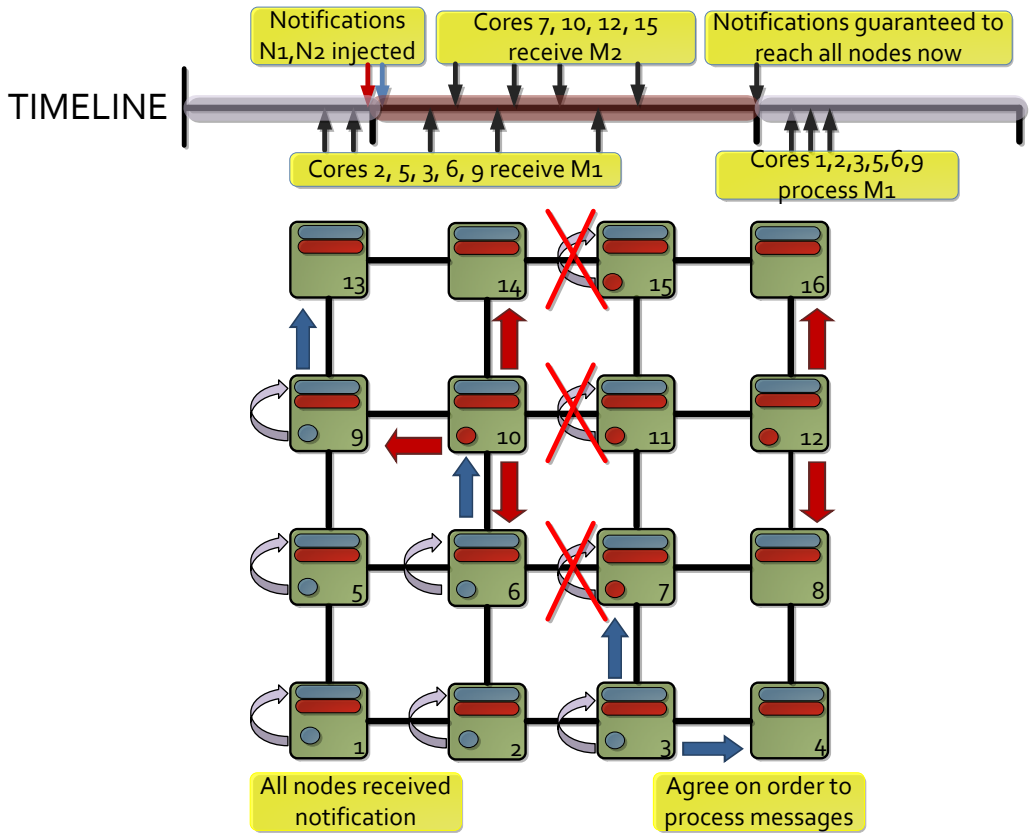


Figure 4-3: All nodes agree on the order of the messages *viz.* $M2$ followed by $M1$. Nodes that have received $M2$ promptly process it, others wait for $M2$ to arrive and hold $M1$

5. At the end of the time-window, we guarantee that all nodes have received any notifications sent during this time window from any node; in this case $N1$ and $N2$ have been received by all nodes in the system. At this instant, all nodes know that they need to process messages from nodes 1 and 11. They take a local decision on how to order these messages. In this case, the rule is "increasing order of SID". Thus all nodes agree to *process* $M2$ *(from node* 1*) before* $M1$ *(from node* 11*)*.

6. If a node has already received $M2$ then it may process the message $M2$ immediately, and subsequently if it has received $M1$ it may process that too. If a node has received neither $M1$ nor $M2$, then it waits for $M2$. If a node has received only $M1$, then it holds the same in the NIC (or the router, depending on availability of buffers) and waits for $M2$ to arrive. The mechanism for the same is as follows. Each NIC main-

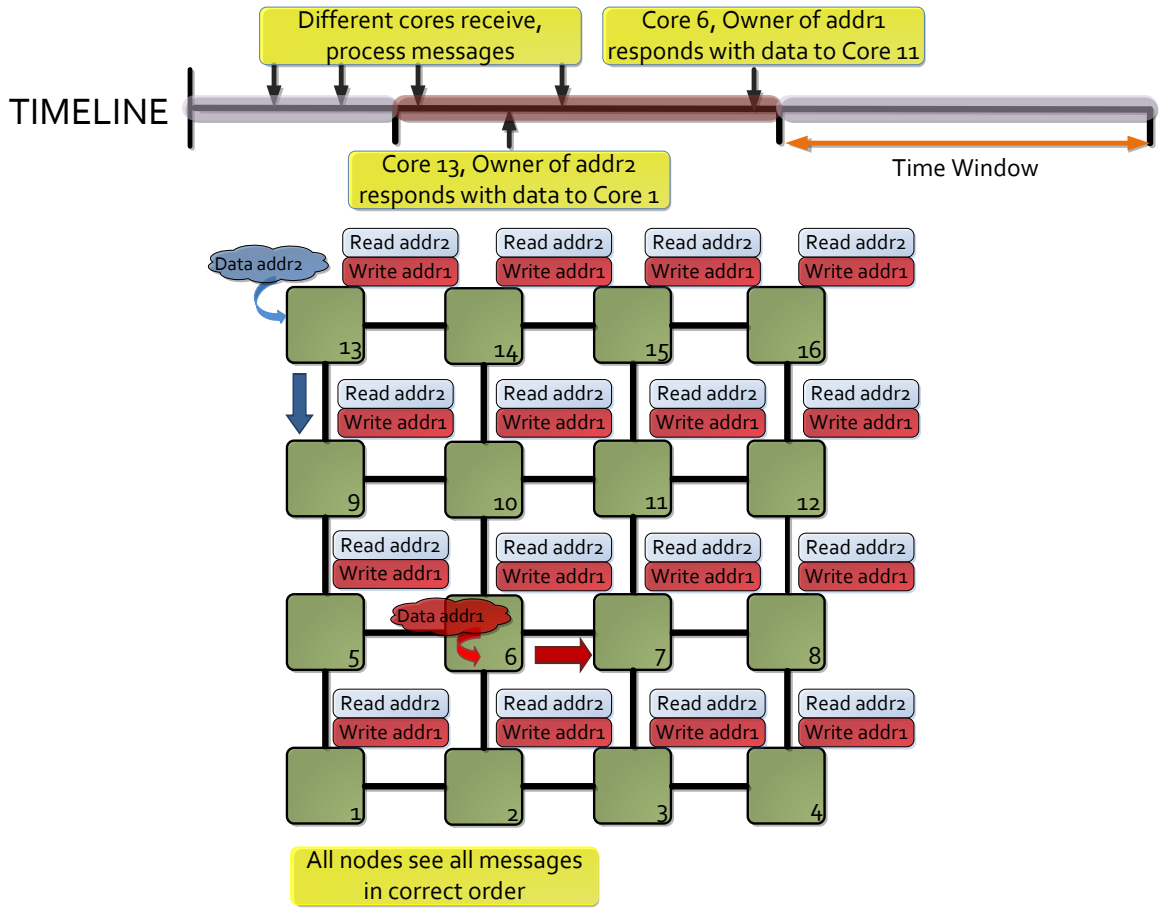


Figure 4-4: All nodes process the messages in the correct order. The relevant nodes respond appropriately to the messages

tains an *expected source ID* (ESID) that represents the source of the next message it must service. When a message arrives on the main network, the NIC performs a check on its SID field. If it matches the ESID, then the message is processed; else it is held in the buffers. Once the message with the SID equal to ESID is processed, the ESID is updated to the next value.

7. Eventually, all the nodes receive $M1$ and $M2$ and process them in the agreed order, namely, $M2$ followed by $M1$.

Figure 4-5 shows the complete timeline of events in the system.

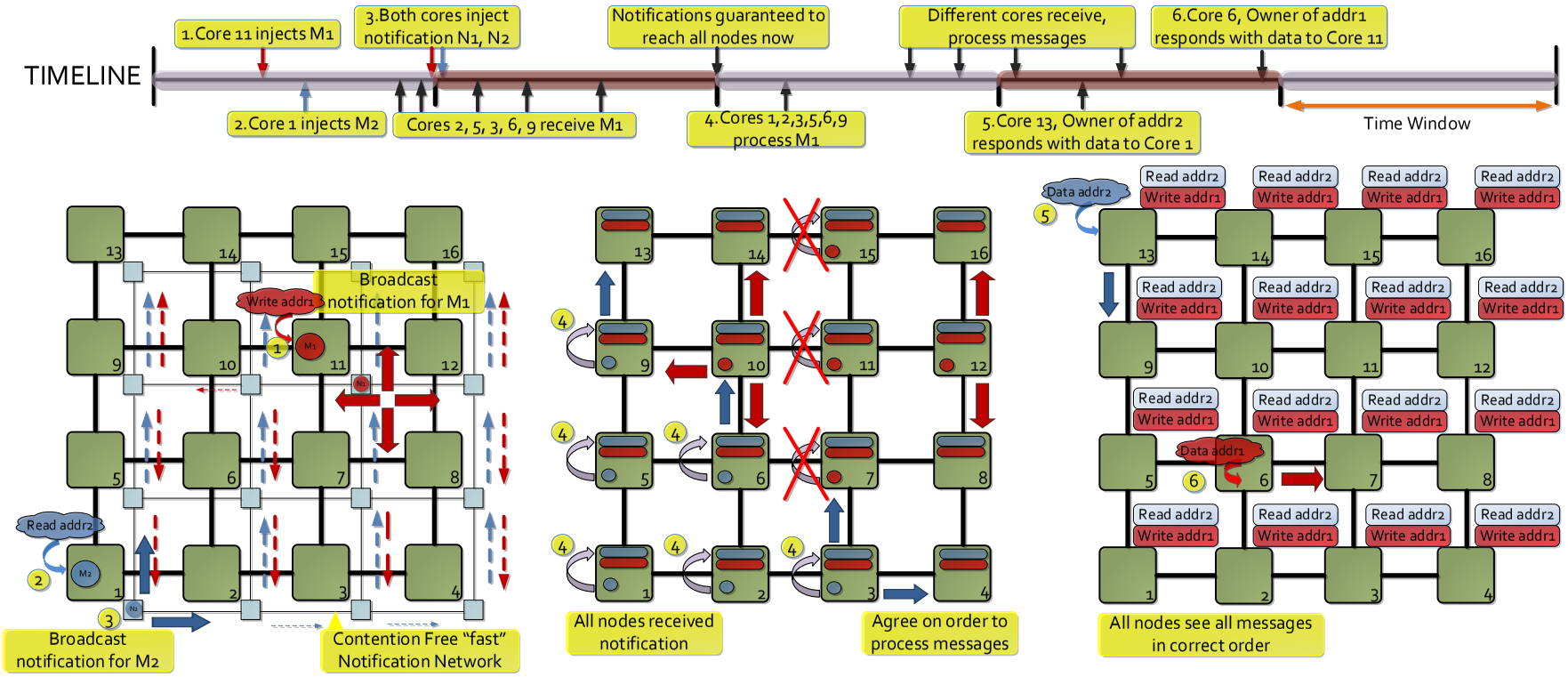


Figure 4-5: Walkthrough example with full timeline of events

4.1.2 Characteristics of OMNI

The walkthrough example in the previous section described the operation of OMNI for a mesh network. However the same scheme may be extended to other network topologies. Here we present a few important general characteristics of OMNI.

1. *Decoupling of message delivery from ordering:* In traditional ordering-point-based interconnection networks, the delivery of the message is tied to the ordering of the message. The message is held up at the ordering node until it has been ordered. In OMNI, we separate these two processes. The actual message may traverse the network and reach its destination at any time. The ordering decisions are separate from this process, and only affect when the NIC processes the messages. While this idea of decoupling the two processes is not new [40], it is beneficial to explicitly identify this separation. We believe that such a decoupling allows for efficient utilization of network resources, and is more scalable.

2. *"Fast" notification network:* In this work, we use a contention-free, lightweight[1] mesh network as the notification network. However, the fundamental requirement of the notification network is the ability to bound the latency of messages. Knowing the latency bound allows us to fix the length of the time window, which in turn clarifies when nodes can send out notification messages and when nodes can make ordering decisions. Further, a separate network is not a strict necessity to enforce a latency bound. But as we demonstrate in section 4.2.2 it is easy to construct a contention-free lightweight network, and provide latency bounds for any system without any need for simulation.

3. *Ordering rule:* In the walkthrough example in section 4.1.1, we used the rule "increasing order of SID" to order the messages. It is possible to define any rule that is a function of the source IDs to order the messages[2]. In the current implemen-

[1] lightweight, in this case, refers to low power and low latency

[2] if we tag the notification with other information, apart from the SID, then it is possible to define ordering rules using that information too; however it could come at increased implementation complexity

tation of OMNI, we use the "increasing SID order" allied with a rotating priority for nodes in each time window. Thus in the first time window, node 1 (node 16) has the highest (lowest) priority, in the next time window node 2 (node 1) gets the highest (lowest) priority, and so on. This ensures fairness in the system.

4. *Notification network – ensuring forward progress:* We note that it is possible to remove the notification network, and simply require all nodes to periodically take a consistent decision on ordering of messages. For example, every node in the system might decide on every time window that it will process messages in the order $1, 2, \cdots, N$. However, if node 1 does not inject any message for a long period of time, then no requests from other nodes can be processed. By sending a notification, we take a decision only on messages that are already in the network, thus ensuring forward progress.

5. *Point-to-point ordering:* Since we are using the SID as the basis for ordering, it is important to ensure point-to-point ordering for messages in the main network. Otherwise, a later message from a given source may overtake an earlier message, and may be incorrectly processed at a destination node. This requirement is only for the message class that the requests travel in.

6. *Deterministic routing requirement:* Apart from point-to-point ordering, deterministic routing is required in the network to guarantee that messages with the same SID do not take different routes. Therefore, in its current form, OMNI does not permit adaptive routing. However adaptive routing may be used for other message classes.

4.1.3 Deadlock avoidance

The OMNI network uses XY-routing algorithm which is deterministic and deadlock-free. While the NIC prioritizes the packet with SID equal to ESID, that alone is insufficient to guarantee that this packet will make forward progress. A deadlock arises in

on-chip networks when packets are stalled because they are unable to obtain a VC or a buffer. Figure 4-6 shows a scenario where packets with SID not equal to ESID hold onto VCs/buffers at the NIC as well as intermediate routers. The NIC, however, is waiting for the packet with SID equal to ESID (= 1), which is blocked at the intermediate router because it is unable to obtain a VC/buffer. In such a scenario the system is deadlocked.

To address this problem, we reserve a virtual channel (VC) and a buffer at every router and NIC, for the packet with SID equal to ESID. This ensures that the packet with ESID always proceeds towards the target without starvation. Figure 4-6 shows how the reserved VC (rVC) provides an "escape path" to the packet, thus guaranteeing deadlock freedom.

4.2 Implementation Details

4.2.1 Router Microarchitecture

Figure 4-7 shows the microarchitecture of the four-stage pipelined router for OMNI. We first explain the broad tasks performed in each pipeline stage, and then explicate the details in each stage.

In the first pipeline stage, the incoming flit gets buffered (BW), and simultaneously arbitrates among other virtual channels (VCs) at its input port for access to the switch (SA-I). In the second stage, the winners of SA-I at each input port arbitrate for using the crossbar switch (SA-O), and simultaneously obtain a free VC (VA) at the next router if available. In the third stage, the flits that won the switch allocation traverse the crossbar (ST) and finally in the fourth stage, the flits coming out of the switch traverse the link to the next router. The router implements single cycle multicasting – allowing flits to be forked through multiple output ports in the same cycle – thus reducing serialization delay.

To reduce traversal latency and buffer read/write power, we employ a lookahead (LA) bypassing scheme [33, 32, 31, 52]. In this scheme, a lookahead containing control information for a flit is sent one cycle prior to the arrival of the flit. The lookahead performs

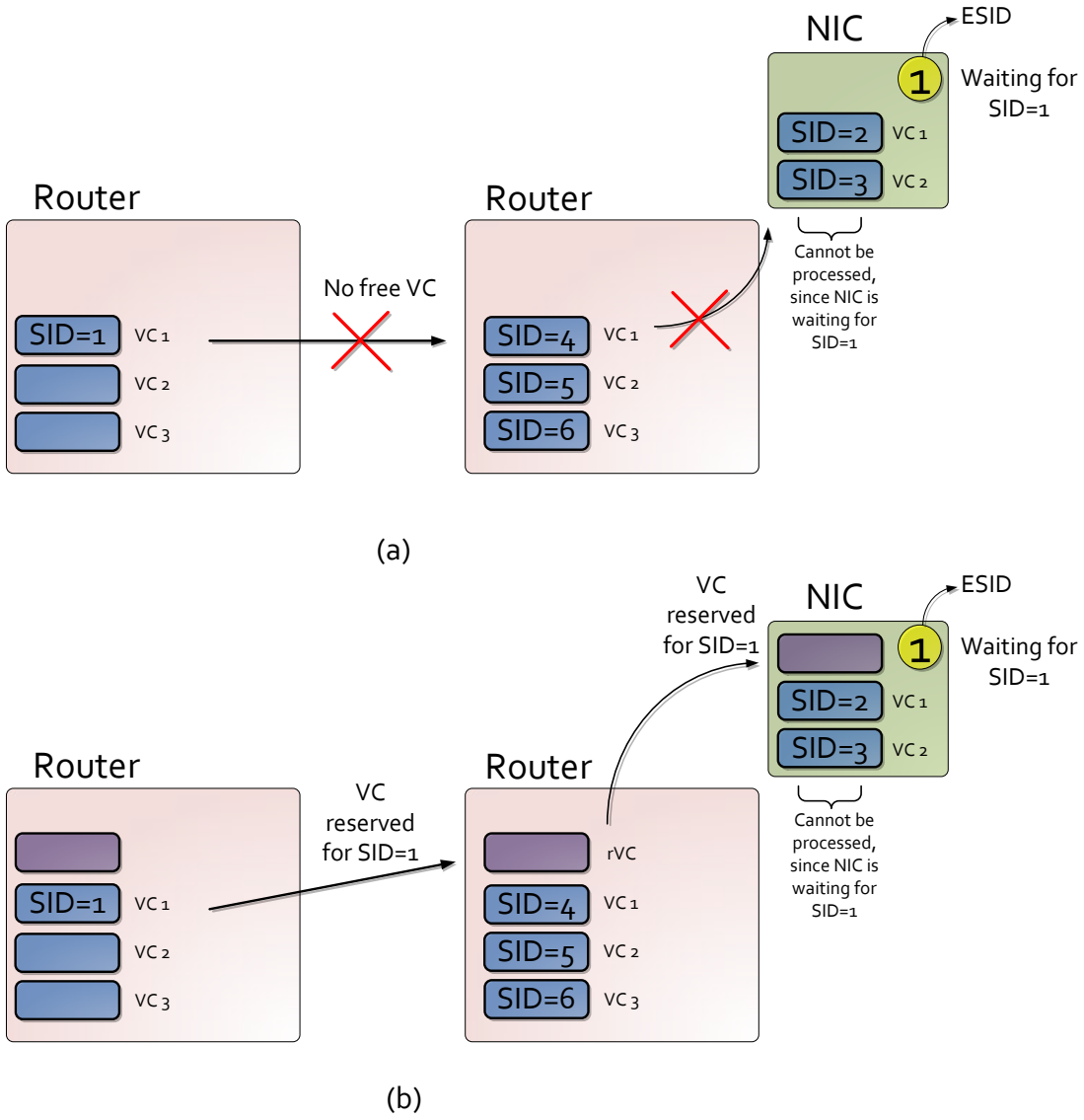


Figure 4-6: (a) NIC waiting for SID=1 cannot process the other flits. Flit with SID=1 is unable to reach the NIC because of lack of VCs (b) Addition of reserved VC (rVC) ensures forward progress

route-computation, and tries to pre-allocate the crossbar for the approaching flit. Lookaheads are prioritized over buffered flits[3] – they attempt to win SA-I, SA-O, obtain a free VC at the next router, and setup the crossbar for the incoming flit, which then bypasses

[3] an exception is made for flits in the reserved VC, which is explained later

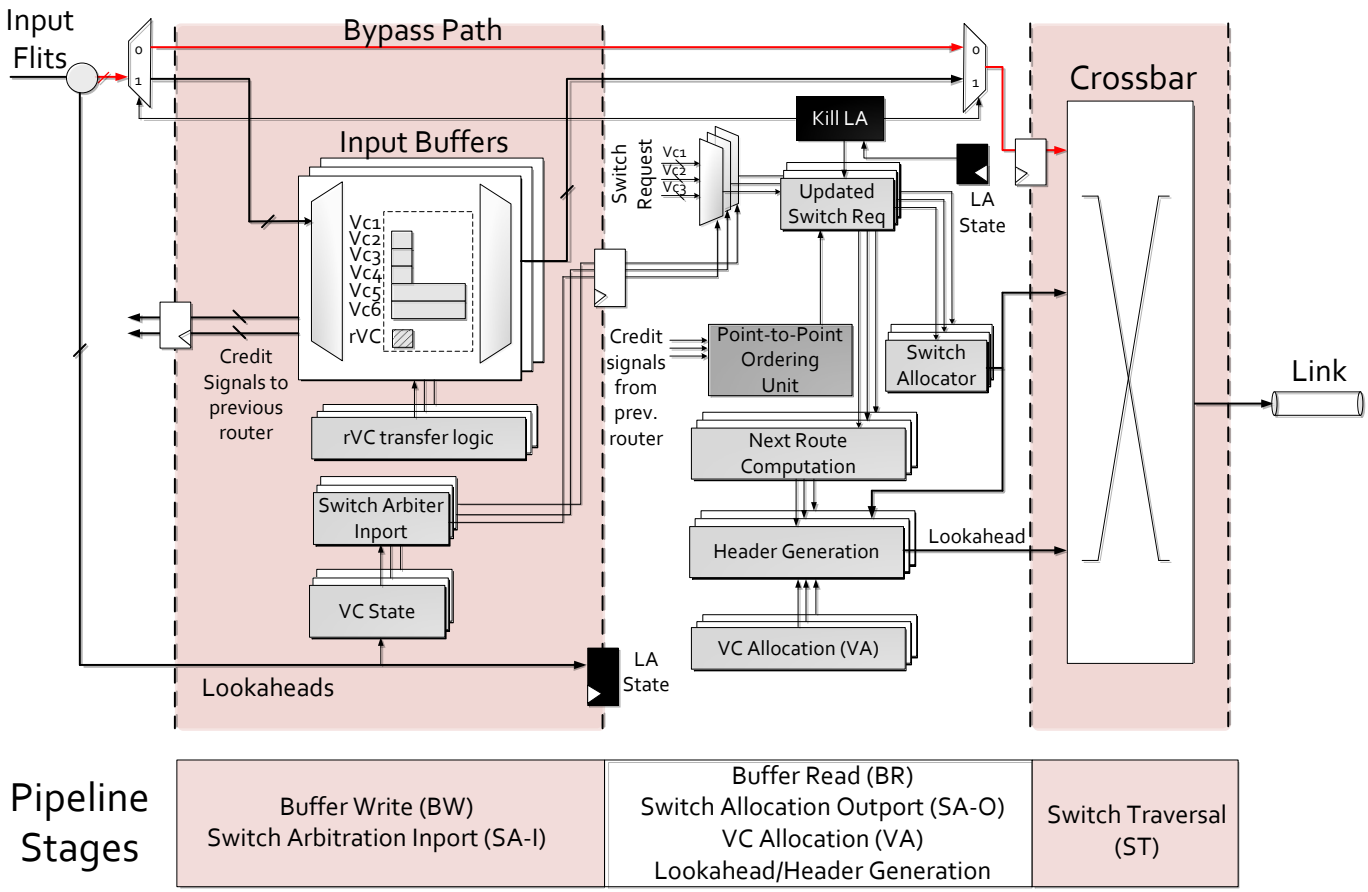


Figure 4-7: Router microarchitecture

the buffer write and moves on to switch-traversal (ST) directly. Conflicts between lookaheads from different input ports are resolved using a static priority. 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 the pipeline stages as in the conventional case. The lookaheads carry information normally included in the header field – destination coordinates, input VC ID and the switch requests – and hence do not impose any overhead. Figure 4-8 shows the regular bypass pipeline and the LA based bypass pipeline. In the bypass pipeline, the LA is sent one cycle in advance and performs LA route compute (LA-RC) and LA-CC (LA conflict check). LA-RC determines the output ports that the flit requires at the next router. LA-CC places a request for the output ports, and check if free VC/buffers are

available at the next router. If a LA is killed, then the flit fall back to the regular non-bypass pipeline. Finally, while the flit performs crossbar traversal (ST), the LA is sent to the next router.

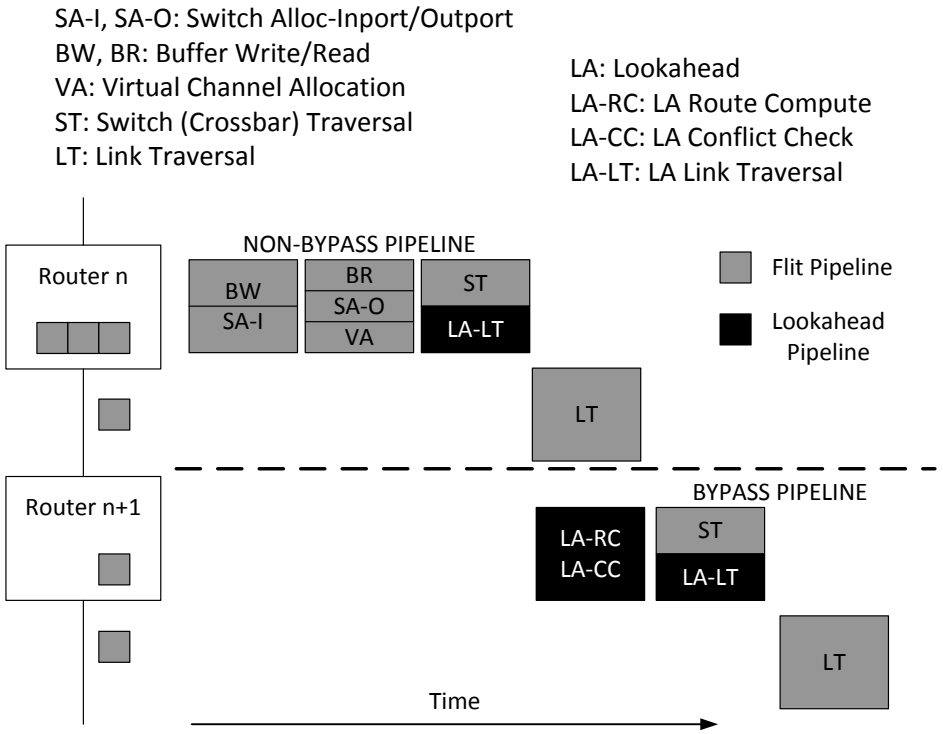


Figure 4-8: Regular and Bypass router pipelines

rVC transfer: As explained in section 4.1.3, we employ a reserved virtual channel (rVC) to avoid deadlock in the network. The NIC at every node maintains an ESID which is passed to the router. The *rVC transfer logic* performs two functions.

1. If an incoming flit has SID equal to ESID, then the rVC transfer logic determines that it should get priority and shifts the flit to the reserved VC. This ensures that the flit gets priority in both SA-I and SA-O.

2. The rVC transfer logic also tracks the SID of the flits that are already buffered in other VCs. When the ESID in the NIC is updated, and if any flit in the buffer has this SID, then it is moved into the rVC to ensure priority access to the switch.

The *VC state* module maintains some important properties of the flits in the buffers such as SID, VC ID and output port request. This allows for the control flow to proceed without having to read from the buffer each cycle.

Point-to-point ordering: There are a few alternatives on how to implement point-to-point ordering. In this implementation, we maintain the following property at each input port to ensure point-to-point ordering of requests in the OREQ network.

> **Property P:** *No two flits at a particular input port of a router, or at the NIC input queue have the same source ID (SID).*
>
> **Claim:** *Property P, coupled with deterministic routing, guarantees point-to-point ordering of all flits between any source-destination pair in the network.*
>
> To see why this is true, 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 the claim holds.

Property P is enforced as follows. For each output port, we maintain a table – referred to as the *SID tracker* table – that tracks the SID of the flit in each VC at the next router. Suppose a flit with SID = 5, wins the north port (N) during SA-O and is allotted VC 1 at the next router in the north direction, then we add an entry in the table for the N port, mapping (VC 1) $\rightarrow$ (SID = 5). At the next router, when the flit obtains all its output ports, a credit signal is sent back to this router. At this point, we clear the entry in the

table. In the intervening period, any flit with SID = 5 is prevented from placing a request to the north port. In the *Updated Switch Req* block, we perform a check on the SIDs in the next router in each direction requested by the flit, and if a match is found we disable the request for that particular direction.

4.2.2 Notification Network

Notifications carry only one information with them, namely the SID. In OMNI, we provide a contention-free notification network as follows. A notification message is essentially a bit-vector of length N. When a core sends out a notification, it asserts the bit corresponding to its SID in the vector. Notifications are broadcast through the network in an XY fashion, similar to messages in the main network. In the event of a contention for a particular port, the contending notifications are combined by performing a bit-wise OR of the messages. The resulting message has the appropriate bits in the bit-vector pulled high, preserving the information in the constituent messages. Each destination node, may receive notifications combined or otherwise, at different points within a time window. Once again, every time a new notification is received, the existing notification is updated by performing a bitwise-OR. At the end of every time window, this bit-vector is read and sent to the network interface controller for processing. In the current implementation of OMNI, we allow only one notification to be sent out per node every time window, so that multiple notifications do not conflict with each other.

The notification network is a light-weight, contention-free network.

1. *Bufferless:* The notification network is bufferless. At each node, in the event of a contention the contending messages are merged and forwarded. This property also allows the notification network to have extremely low power overhead.

2. *Single cycle per hop:* Since there is no contention, notification messages traverse a hop in a single cycle.

3. *Latency bound:* In OMNI, since all notification messages are injected at the start

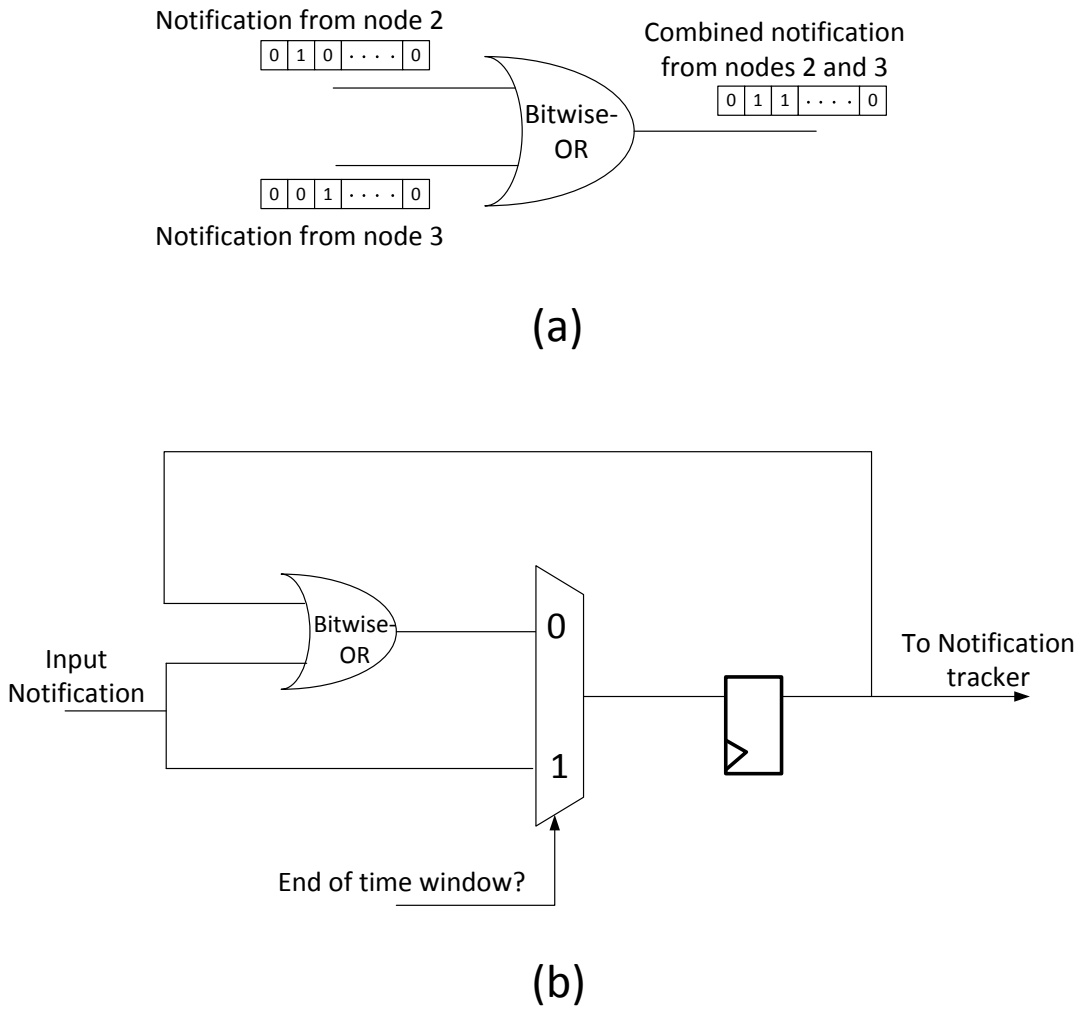


Figure 4-9: (a) Notifications may be combined using bitwise-OR, allowing for a contention-free network (b) We aggregate incoming notifications through the time window. At the end of the time window, it is read by the notification tracker.

of a time window, they are guaranteed to be delivered to all nodes in the system in time $T < T_{bound}$, where T_{bound} = (Number of X nodes + Number of Y nodes) $= 2\sqrt{N}$.

4.2.3 Network Interface Controller Microarchitecture

Figure 4-10 shows the microarchitecture of the OMNI network interface controller. The network interface controller (NIC) provides an AMBA ACE interface to the L2 cache

controllers. It accepts requests and encapsulates them into network packets and flits. It determines the virtual network the packet must be sent out of and inserts into the appropriate queue. For packets in the *Globally Ordered Request* virtual network, the NIC handles sending out notifications at the beginning of time windows. It also participates in maintaining point-to-point ordering of messages in the network. On the input side, the NIC accepts and tracks notifications from various nodes through the notification tracker module. It communicates the ESID to the arbitration unit, which ensures global order for packets in the *Globally Ordered Request* virtual network. Finally, packets are parsed, and appropriate information passed to the L2 cache controller through the AMBA ACE interface.

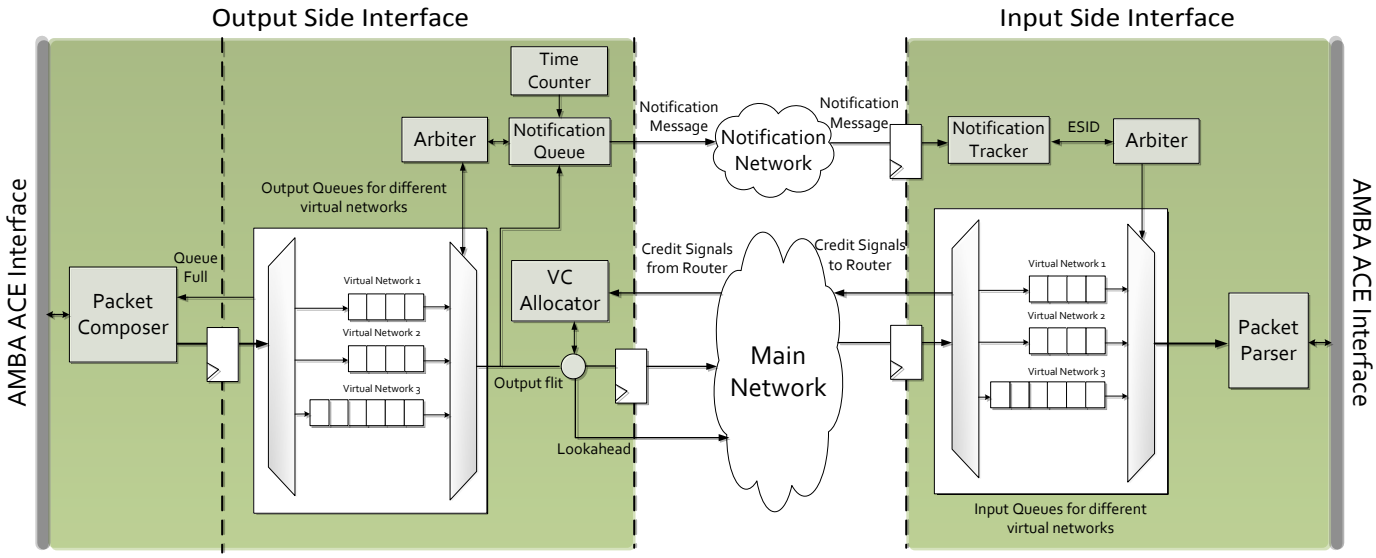


Figure 4-10: Network Interface Controller microarchitecture

Notification Queue: The notification queue sends out a notification for every message that has been injected into the main network. The notification is injected at the start of every time window. In the current implementation, we allow only one notification to be sent out every time window. As explained in the previous chapter, this is because the cores in the SCORPIO processor can have at most two outstanding requests at any point

in time. However, if required (say to handle more bursty traffic), it is possible to group notifications and send them out in the same time window. Note that the restriction of one notification per time window does not preclude sending out multiple messages in the main network in a single time window. If multiple messages are sent out, then we send out the corresponding notifications in the future time windows. This simply involves maintaining a counter for the number of notifications that must be sent out. If the counter is greater than zero at the start of the time window, then we send out a notification and decrement the counter. If the counter reaches the maximum value, then we throttle the messages being injected into the main network.

Notification Queue Operation

```
1:  NullifCounter ← 0
2:  if Message sent out on main network then
3:      NotifCounter ← NotifCounter +1
4:  end if
5:  if (Beginning of time window) AND (NotifCounter > 1) then
6:      Send notification
7:      Decrement NotifCounter
8:  end if
9:  if (Counter == MAX) then
10:     Stall main network flits
11: end if
```

Notification Tracker: The notification tracker processes notifications received from different nodes. It is responsible for maintaining the ESID value which the arbiter then uses to process messages in the *Globally Ordered Request* queue in the correct order. The notification tracker reads aggregated notification bit-vector from the notification network at the end of every time window. The bit vector represents the set of nodes to service next, and is referred to as a *notification entry*. A notification entry is taken up for processing immediately, unless there are prior notification entries that still need to be processed, in which case, the notification entry is entered into a *notification entry store* for future processing. A notification entry is passed through a priority arbiter to determine the ESID

– the highest priority bit in the bit vector is the ESID for this node. When a packet with ESID is processed, the corresponding bit in the notification entry is de-asserted and the priority arbiter determines the next ESID. When all the asserted bits in the notification entry have been processed, a new notification entry from the notification entry store is taken up for processing if available. Else, the notification tracker waits until the next time window to obtain a new notification entry. The rotating priority is also updated at this stage.

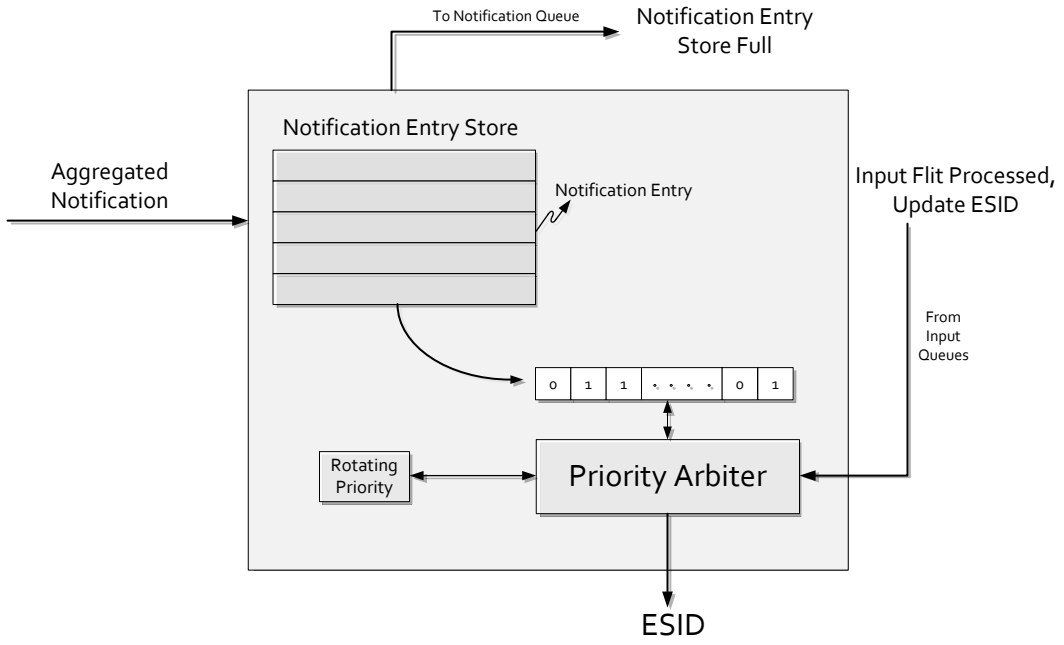


Figure 4-11: Notification Tracker Module

4.2.4 Few other implementation details

There are a several subtle issues involved in the implementation of OMNI. We present few of them here.

Credit signaling for the reserved-VC: Credit signaling for the rVC presents two specific issues that need to be handled.

1. We only have one set of credit-signaling links between routers. The rVC transfer logic allows for a flit to be moved from a regular VC to a rVC. In such a scenario, we need to signal to the previous router that the regular VC has been freed and the rVC is now occupied. These signals cannot be sent in different cycles, since the previous router takes a decision on VC allocation on a cycle-by-cycle basis. This issue assumes greater importance in the context of the point-to-point ordering logic in the network, which is fundamentally connected to the credit signaling mechanism – credit signals indicate when a VC ID → SID map may be cleared in the SID tracker table. Hence, we add an additional bit to the credit signal that indicates when a move has occurred. If a credit is received, and the move bit is set high, then it is implied that the rVC is now occupied. In addition, the SID originally in the regular VC is now mapped to the rVC instead.

2. The single credit-signaling link throws up another issue. We could encounter a scenario where a flit in VC A is moved to the rVC, and in the same cycle, a flit in a different VC B could win access to all its output ports. Then, we have two virtual channels (VC A and VC B) that become free in the same cycle, but it is not possible to send a credit for both these VCs. While there are a few possible solutions to this, it is important to be cognizant of this possibility and account for the same in hardware. In the current implementation of OMNI, in such a scenario, the move into the rVC is prioritized, and we kill the grant to exactly one port for VC B[4] – namely the grant to the local port. This is because, the possibility of a move to the rVC suggests there is a higher priority flit in the input port. Hence we delay the grant to the flit in VC B, prioritizing the flit in the rVC instead.

ESID and interactions with rVC: The interaction of the ESID with the rVC moves and credit signaling, in the context of a latency-sensitive design, provides for interesting

[4] we only need to ensure that VC B does not win access to all the output ports; the flit may still traverse other output ports it is granted

corner cases that must be handled with care.

1. *Neighboring routers ESID variation:* A key characteristic of OMNI is that different routers in the system can progress at different rates while processing messages in the correct global order. As a result, two different nodes might have different ESIDs. It is important to ensure that the reserved VC at a node is allocated to a flit with the same SID as the ESID of that node. This is important because VC allocation is typically done upstream at a different router. It is also possible to have ESID sequences such as $\{\cdots,7,6,7,\cdots\}$ – two different routers might have ESIDs corresponding to the two different 7s. It is important to employ suitable checks to ensure correct allocation of the rVC to prevent deadlock[5].

2. *Initialization of ESID and impact on rVC:* On start-up, the ESID values in the system are initialized to `0`. However, depending on when the first packets of the system are injected relative to the time window, `0` may not be the first packet in the global ordering. It is possible that a flit with SID $=$ `0` incorrectly occupies the rVC and leads to a deadlock. Hence, we disable the rVC at all nodes during the initialization phase. Once the first set of notifications have been received, the ESID is set to a valid value and then the rVCs throughout the system are enabled. This ensures correct and deadlock-free behavior.

Notification Entry Store Overflow: The notification entry store effectively holds the decisions on global ordering at each individual node. Since different nodes may progress at different rates, it is possible that the notification entry store at one node becomes full. In this scenario, we must halt the notifications sent from any other node, until there is space in the notification entry store. To effect this, the node in question, sends out a *STOP* notification – this is simply one additional bit in the notification message bit-vector. On receiving a *STOP* notification all nodes cease sending notifications i.e. the

[5]if we are not careful, the second 7 might occupy a rVC while the first 7 is being processed, and later the ESID might change to 6 and flit 6 may be starved of resources causing deadlock

Notification Queue module is throttled. Once the affected node makes progress, releasing space in its notification entry store, it sends out a *START* notification to all nodes, at which point, the network reverts back to normal operation. Here again, we note that this does not preclude sending out messages on the main network. We point out that multiple nodes may have full notification entry stores at the same time, and they would send out *STOP* notifications at the respective time windows. However, they may make progress at different rates, and therefore, one of the nodes might send out a *START* notification even while other nodes still have a full notification entry store. Tracking the *STOP* and *START* notifications for every node is tedious and expensive in hardware. Instead, we require that nodes with a full notification entry store send *STOP* notifications every time window until it makes forward progress, freeing up space in its notification entry store. Since *STOP* notifications assume higher priority, they override any *START* notifications. Also since all nodes are guaranteed to make forward progress, eventually atleast one of the affected nodes sends out an effective *START* signal bringing the network back to normal operation. We note that by suitably sizing the notification entry store, this event can be made sufficiently rare. Further, since the notification network is lightweight, this mechanism does not impose any significant power overhead.

Point-to-point ordering for the *P2P-REQ* network: The *P2P-REQ* network handles non-cacheable reads and writes which require point-to-point ordering. Since this traffic is typically low load, we implement a simpler scheme to enforce point-to-point ordering of requests in this virtual network. We employ a queueing arbiter for this virtual network alone, which prioritizes earlier packets to later ones. Coupled with deterministic routing, this scheme guarantees point-to-point ordering for requests on this virtual network.

4.3 Chapter Summary

This chapter proposes an in-network distributed ordering scheme, to enable snoopy coherence on unordered interconnects. The network maintains global ordering by requiring all nodes to agree on the ordering of messages in the network at synchronized time intervals. The nodes reach this agreement through a local decision, based on notifications they receive about approaching network messages. By ensuring that all nodes see the same set of notifications at these synchronized time intervals, and performing a consistent local decision at each node, we guarantee global ordering of all messages. This chapter describes how this scheme may be realized in a mesh interconnect called OMNI. We present details on the router and network interface controller (NIC) microarchitecture for OMNI, including subtle issues that arise in the implementation.

5 In-Network Filtering in OMNI

Broadcasting of messages is a key characteristic of snoopy coherence. This is because snoopy coherence does not maintain sharing information, as is done in a directory protocol. As we scale to a greater number of cores in the CMP era, broadcasting all snoop requests may have substantial network bandwidth and power implications. The OMNI router employs microarchitectural techniques such as single-cycle multicasting, and lookahead-bypassing to minimize network latency in the presence of broadcast traffic. However, there are architectural techniques that promise further savings.

In-network coherence filtering (INCF) [8] proposed to embed small coherence filters at every router to dynamically track sharing patterns among various cores and filter away *redundant snoop requests*[1], thus saving network bandwidth and power. In this chapter, we explore how INCF may be adapted to OMNI and the microarchitectural changes required to support the same.

5.1 Overview

There have been several snoop filtering proposals in literature [13, 41, 42, 44]. They all involve tracking the cache lines shared at various caches, and filtering requests that will

[1] A redundant coherence request is one that reaches a destination core that does not share the cache line being snooped, thus unnecessarily consuming resources

definitely miss in a particular cache. Conventionally, information about cache coherence is maintained on a per-block granularity. However on-chip storage is precious, and hence prior works have proposed tracking of sharing information at a coarser granularity than cache lines, using *regions*. A region is contiguous portion of memory addresses and each physical cache line maps to exactly one region. It has been observed that if a cache line is not shared among a set of cores, then there is a high probability that the region it belongs to is also not shared among the same set of cores. Thus tracking information at the region granularity suffices for the purposes of filtering.

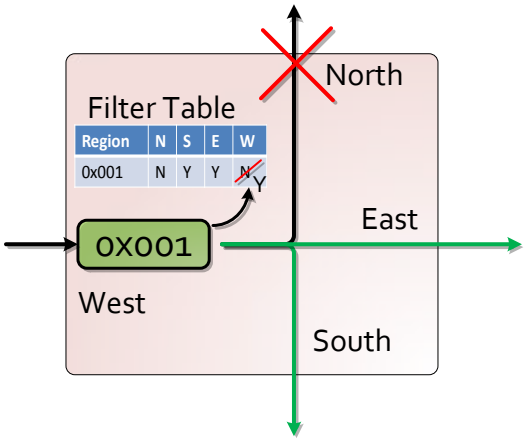


Figure 5-1: Conceptual image of router filtering away redundant snoop request

We maintain region-granularity filtering information at each router port through *filter-table* structures. Specifically, the filter table tracks the regions that are not shared by any of the cores that can be reached from that output port. The router filters away requests to those regions along that particular direction, saving interconnect power and bandwidth. The filter tables are populated as broadcast requests from nodes traverse the network. A broadcast request from a particular port indicates that a core in that direction intends to share a particular region. Therefore, the router clears the entry for the region in that direction in its filter table. When a remote cache does not share a particular region, it informs its local router. The router checks to see if an entry must be added to the filter table and takes appropriate action. It then propagates this non-sharing information to neighboring routers, which in-turn forward the non-sharing information

to their neighbors in an appropriate manner. Thus, over a period of time non-sharing information spreads through the network, and the network becomes rich with filtering information. This allows subsequent requests to a region to be filtered away.

5.2 Implementation Details

5.2.1 Sharing and Non-Sharing Updates

Each cache controller implements a region tracker [53] that maintains information on whether a region is being shared or not. When the cache encounters a cache miss and a region miss, then a sharing update needs to be injected into the network. Similarly, when a region is not being shared, a non-sharing update message is injected into the network. Since these update messages do not have to be globally ordered, we reuse the *Point-to-point Ordered Request* virtual network in OMNI to carry them.

Non-sharing updates are easy to handle. The cache controller injects a message indicating that a particular region is not being shared. The NIC then encapsulates this message into a single flit packet, and injects the same into the *Point-to-point Ordered Request* virtual network. Non-sharing update messages are propagated through the network in accordance with the pseudo-code shown below. The propagation of the update message is tightly coupled to the routing algorithm, since it dictates which nodes are reachable from a particular output port. Since OMNI uses XY routing for globally ordered request messages, the update messages follow a YX route in the network. Each router on the path, takes an appropriate decision as per the pseudo-code.

Sharing updates require careful handling. On a region miss, the cache controller simply injects a globally ordered miss request into the NIC along with a single bit indicating that this request will trigger the sharing of a new region. In coherence protocols, before broadcasting a request into the network, the cache line goes into an intermediate state and the core needs to observe *all* subsequent requests to this line. However the update messages in the network are not instantaneous. This might lead to subsequent

Non-Sharing Region Update

Require: If no router exists in a direction, set the corresponding port bit

```
1: if NIC, North, South and East port are set then
2:     Notify West port
3: end if
4: if NIC, North, South and West port are set then
5:     Notify East port
6: end if
7: if NIC and South port are set then
8:     Notify North port
9: end if
10: if NIC and North port are set then
11:     Notify South port
12: end if
```

Sharing Region Update

```
1: if NIC is set then
2:     Notify all ports
3: end if
4: if South port is set then
5:     Notify North, East and West ports
6: end if
7: if North port is set then
8:     Notify South, East and West ports
9: end if
10: if East port is set then
11:     Notify West port
12: end if
13: if West port is set then
14:     Notify East port
15: end if
```

requests from other cores being filtered away, which can lead to incoherence. In practice, remote requests ordered before the core's own request can be ignored. This is because the requesting core does not start owning/sharing a line until it sees its own request in global order. We use this fact to stall the actual request until the sharing information has been propagated to all nodes. In addition, the request needs to be ordered behind any messages to the region that may potentially have been filtered in the intervening period. We recognize that any messages in the network, are tied to corresponding notifications. Therefore, ordering this request behind other requests in the network boils down to the notification for this request reaching nodes after all the notifications tied to the messages in the network.

When a request requires a sharing update, the NIC moves the request to a *stall buffer* and injects a sharing update for the region into the network. It holds the request in the stall buffer until it receives an acknowledgement that the update has been seen at all nodes. Because of the YX routing path that sharing updates follow, we do not require an acknowledgement from every node, rather an acknowledgement from the boundary nodes along the east and west edges of the network suffice – we call these nodes as the *edge nodes*. As the sharing updates propagate through the network, they update the filter table in the routers. On reaching an edge node, the corresponding router triggers an acknowledgement for the region and sends it to the source of the sharing update. To prevent deadlock, the acknowledgement is sent on a separate virtual network – in OMNI we reuse the *Unordered Response* virtual network, which supports unicasts.

Once the source node receives all the acknowledgements, we still need to guarantee that the request is ordered behind any other requests in the network. As mentioned before, this involves determining when to send out the notification for this request. We use the fact that the notification queue at each node has a finite depth – say D_{NQ}. Any message in the network belongs to one of the following classes.

- The message has been ordered already by virtue of its notification being broadcast and received by all nodes. Therefore, regardless of whether it was filtered or not, it

has already been ordered.

- The second class of messages are those that are in their network, but their notification is yet to be sent out, because the notification has been held at the notification queue of their source. Such messages could potentially get filtered and the request needs to be ordered behind such messages. This is guaranteed if the NIC waits for D_{NQ} time windows to send out the notification after receiving all the acknowledgements. Once all the acknowledgements have been received, no more messages will be filtered, and hence the relative order of those messages is irrelevant.

Thus, we enforce that a stalled request, upon receiving all sharing update acknowledgements, wait for an additional D_{NQ} time windows before its notification is injected. We also choose to send out the actual request at this stage. Naturally, this process incurs a latency penalty for this request. However, region requests are sufficiently rare that this should affect performance negligibly.

5.2.2 Nullification Messages

While messages are filtered within the network, notifications for these messages are still sent out to all nodes. Therefore, all nodes wait for the corresponding messages to arrive to satisfy the global order. To ensure that this does not result in a deadlock or incorrect behavior (a second request from the same source may be mistakenly processed if the first request was filtered away), we send short nullification messages for every filtered message. The nullification message is sent only on the ports that flit was filtered away, and is constructed similar to a notification message – as bit-vectors – and sent through a separate contention-free lightweight network, referred to as the *nullification network*. The nullification message cancels out a notification from the particular source. We describe how this cancellation is effected in section 5.3. Nullification messages also follow the XY routing protocol.

One issue with sending out nullification messages is that they may violate the point-

Stalled Request Operation

```
1:  Initialize: EXPECTED_ACKS = 2√N
2:  Initialize: Okay_to_send = 0
3:  Initialize: TIME_TO_WAIT = D_NQ
4:  Initialize: NUM_TIME_WINS = 0
5:  Initialize: NUM_ACKs_RECVD = 0
6:
7:  if ACK received then
8:      NUM_ACKs_RECVD = NUM_ACKs_RECVD +1
9:  end if
10:
11: if (NUM_ACKs_RECVD < EXPECTED_ACKS) then
12:     Okay_to_send = 0
13: else
14:     Okay_to_send = 1
15: end if
16:
17: if (Okay_to_send ≠ 0) then
18:     while (NUM_TIME_WINS ≤ TIME_TO_WAIT) do
19:         Stall Request
20:     end while
21:     if (NUM_TIME_WINS > TIME_TO_WAIT) then
22:         Send Request
23:         Reset NUM_ACKs_RECVD, NUM_TIME_WINS, Okay_to_send
24:     end if
25: else
26:     Stall Request
27: end if
28:
29: Always @ End of time window
30:    NUM_TIME_WINS = NUM_TIME_WINS +1
```

to-point ordering requirement in the *main network*. This might happen if a nullification message overtakes a main network message with the same SID. Since nullification messages only carry the SID, if a nullification message overtakes a main network message with the same SID, then it will cause incorrect updation of the ESID at destination nodes. We need to disallow such overtaking, while also ensuring that the nullification is received by the destinations. As nullifications traverse the network, at each router, we perform a check to see if any buffered flits have the same SID as the nullification. If yes, then the nullification is merged with the flit in the main network. Since this could happen multiple times, we append a *piggyback counter* to every flit. Whenever a nullification is merged, the piggyback counter is incremented. At the destination NIC, when the flit is processed i.e. it has been ordered and is ready to be parsed and sent to the cache controllers, we read out the piggyback counter. The next time the NIC ESID reaches this value, we know that the corresponding flit has been filtered away. We update the ESID and decrement the piggyback counter for the SID. We note that the number of bits required for the piggyback counter is small and bounded above by the maximum number of hops a flit needs to traverse – namely, (Number of X nodes + Number of Y nodes) $= 2\sqrt{N}$.

5.3 Microarchitectural Changes

5.3.1 Router Microarchitecture

The microarchitecture of the filtering router is shown in figure 5-2. It employs the same four-stage pipeline as the OMNI router, and includes single-cycle multicasting and lookahead-bypassing.

Nullification Merge: This module tracks the SID of buffered flits in various VCs, and compares them against incoming nullifications every cycle. In the event of a match, the piggyback counter for the flit is incremented, and a mask is generated for the correspond-

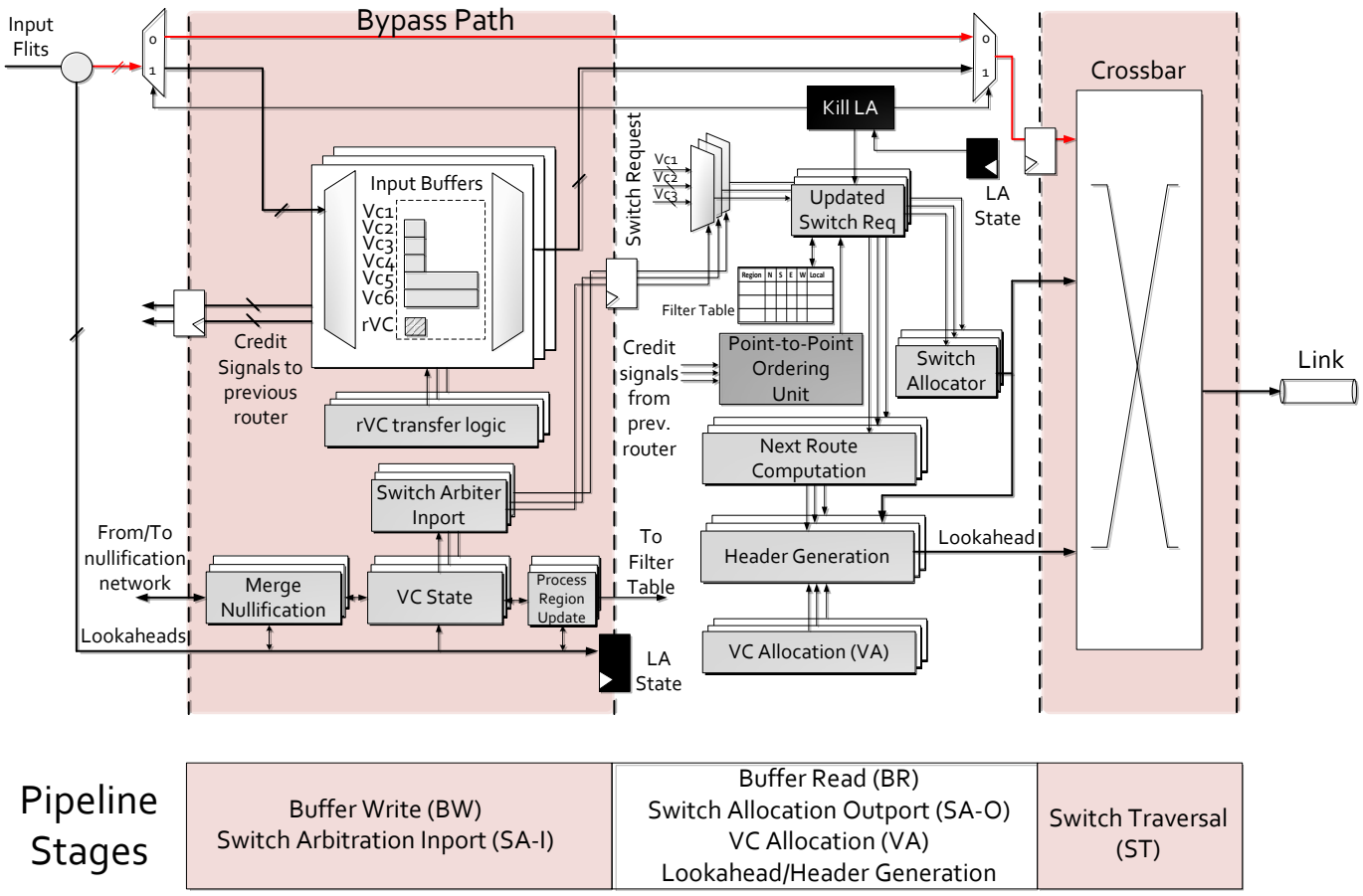


Figure 5-2: Router microarchitecture

ing bit in the nullification message.

Process Region Update: Sharing and non-sharing update messages are processed by this module. It determines whether any change is required in the filter table. It also returns the output ports the update message must be forwarded out of, in accordance to the pseudo-code discussed in section 5.2.1.

Filter Table: As the name indicates, this module maintains the filter table for each output port. The filter table maps regions to non-sharing information along different directions. We refer to each entry as a *filter row* – it is essentially a 5-bit vector with a 1 in-

dicating that the region is not shared along that direction. The filter table need not store filter rows for every region, rather it serves as a cache. Regions that have been queried recently find a place in the cache, while older entries are swapped out. This saves precious storage space and improves cycle time as well. Each input port queries the filter table every cycle with a region, and the filter table responds with the filter row if it exists. Each input port may also seek to update the filter table, in the event of a sharing or non-sharing update message. The filter row returned by the filter table is used to filter away request flits at the input port. If a flit is filtered along a particular direction, then a nullification message is generated for that direction and sent into the nullification network.

5.3.2 Network Interface Controller Microarchitecture

Figure 5-3 shows the microarchitecture of the network interface controller. It is quite similar to the OMNI NIC, but has a few additional components.

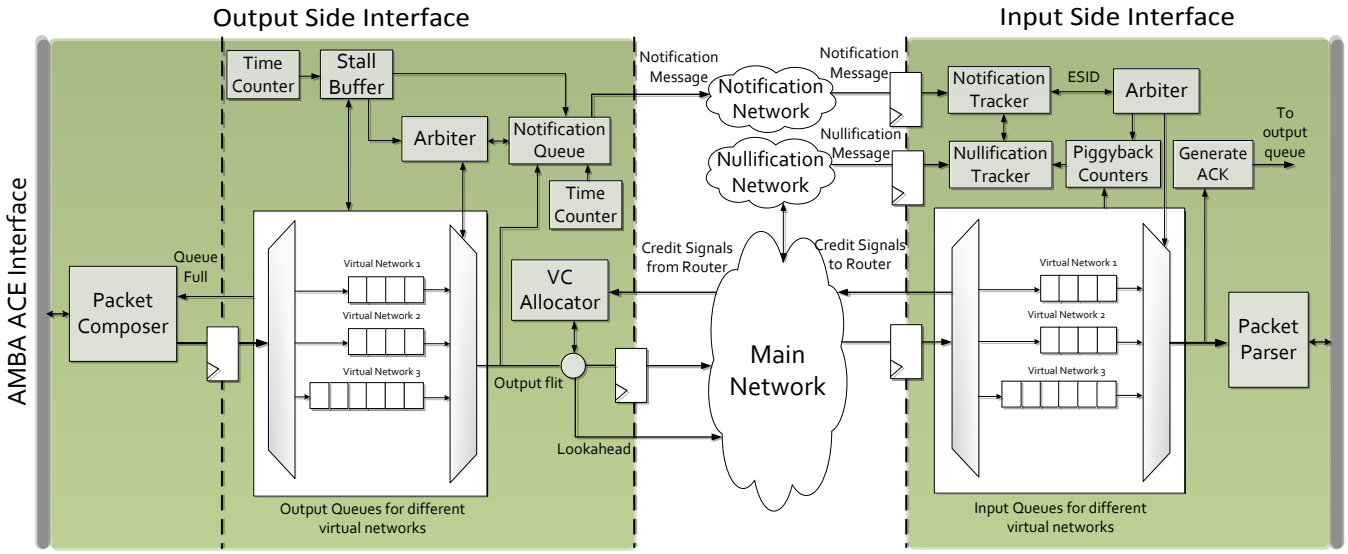


Figure 5-3: Network Interface Controller microarchitecture

Stall Buffer: The stall buffer holds flits that require a sharing update. It gathers the ACKs received for each stalled flit and holds the flit for the required number of time win-

dows. It indicates to the arbiter and notification queue, when the flit and notification are ready to be sent out. The input side interface of the NIC communicates with this module whenever an ACK is received.

Nullification Tracker: The nullification tracker essentially maintains a counter for each SID, tracking the number of nullifications received for that SID. If the ESID has a non-zero nullification count, then the ESID may be safely updated to the next value while decrementing the nullification count. We refer to the counters within the module as *nullification counters*.

Piggyback Counters: The piggyback counters module reads the piggyback counters in incoming flits. Once the flit wins the arbitration, the piggyback counter may be safely added to the nullification counter.

Generate ACK: This module is present on the *edge nodes*, and generates an ACK for sharing updates. It determines the route for the unicast flit depending on the SID of the sharing update. It checks that there is sufficient space in the *Unordered Response* output queue, and inserts the ACK.

5.3.3 Few other implementation details

As in the case of OMNI, filtering also presents subtle implementation issues, a few of which are discussed here.

Updating filter table: Multiple input ports can try to update the same filter row in the same cycle. This needs to be taken into account while updating the filter row. In the simplest case, the different input ports attempt to update the same region. However since the filter table only serves as a cache, we could have updates to different regions that map to the same index in the cache. In such cases, we can only have one unique region in that

line of the filter table. Here we resolve this case using a static priority among the input ports. It is possible to use more refined rules such as favoring non-sharing information over sharing information, since non-sharing information allows us to filter away requests. However such scenarios are relatively rare even for small filter table caches. Hence we use a simple fixed priority.

Filtering flits with piggybacked nullifications: Flits with piggybacked nullifications are not filtered in the current implementation. This is because, the corresponding nullification message should indicate it is carrying two nullifications. While it is possible to augment nullification messages with a counter, we disallow filtering of flits with piggybacked nullifications for simplicity.

Nullification counter, ESID and rVC: A nullification counter might trigger an ESID update. However the ESID update may not happen instantaneously. This might be because the next notification has not yet been received, and so we do not know what the next ESID value should be. However, this does not preclude having a main network message with the same SID. It is important to disallow transfers to the rVC in the router in the intervening period. The ESID valid signal used in OMNI, is revised to include this check too. The router always checks the valid signal before initiating an rVC transfer.

Nullification tracker, piggyback counters and flits: In the input side interface, we always give priority to the nullification tracker to trigger an ESID update. If a flit with the SID equal to ESID is available in the input queues, and the nullification counter is non-zero, then the nullification counter is the one that triggers the ESID update. What this means is that the nullification was received before this flit, and hence it must update the ESID and maintain the correct global order. The flit will be processed when the ESID next comes around to the same value.

Now for incoming flits with piggybacked nullifications, the flit needs to be ordered

first, and then the nullifications it is piggybacking. Hence we hold the piggybacked nullifications in *Piggyback counters* until the flit is processed, and then update the counters in the *Nullification Tracker* module.

A third scenario is when a nullification is received, followed by a flit with the same SID, followed by more nullifications. In this case, the second set of nullifications should be handled after the flit has been processed. Once again, the second set of nullifications update the piggyback counters. In this scenario, first the nullification updates the ESID, followed by the flit in the input queue. Then the piggyback counters update the counters in the nullification tracker module. This ensures point-to-point ordering and in-turn correct global ordering.

Nullifications overtaking notifications: A seemingly benign question is when the piggyback counters should update the counters in the nullification tracker module. It appears that once the flit with corresponding SID has been processed, it would be safe to update the nullification counters. However, while processing of a flit triggers an ESID update, there may be delay in actually updating to the next ESID. For example, there are no more notifications to process in notification tracker, and hence the next ESID is not known. The issue here is that nullifications can overtake the notifications they were intended to cancel – just as flits can arrive at destination nodes before their notification, so too can nullifications. And just as flits are held in the input queues until they are ready to be processed, so too should we hold nullifications in the event that the notification has not arrived.

In the simple case, the nullification arrives before the notification and updates the nullification counters. Only when the notification arrives do we trigger the ESID to be updated to the corresponding SID. Then rather than the flit triggering the ESID update, the nullification safely triggers the update.

With piggybacked counters, the nullification counters are not updated immediately upon processing of the flit in the input queue. Instead, we wait for the ESID to be updated

to the next value and then update the nullification counters. If not, since the ESID stays at the same value for multiple cycles, we may accidentally trigger additional ESID updates and incorrectly decrement the nullification counters. This check may be performed using the revised valid signal for the ESID.

5.4 Chapter Summary

This chapter builds on prior work on in-network filtering, and describes how such filtering mechanisms may be enabled in OMNI. We describe the changes to the network microarchitecture, and also point out subtle implementation issues and how to tackle them.

6 Results and Evaluations

The SCORPIO processor has been fabricated in a commercial 45 nm technology, and includes a 6x6 mesh interconnect OMNI that connects the 36 cores and enables snoopy coherence among them. This chapter presents architectural evaluations that demonstrate the advantage of OMNI in comparison to other NoC implementations. This is followed by a discussion on validation of the network RTL, and the performance characteristics of the network. We then present the post-synthesis and layout results, and power characteristics of OMNI. We also present analysis of the filtering optimizations on OMNI, and the rationale for not including some of these optimizations in the SCORPIO prototype.

6.1 Full System Evaluations

We perform full-system simulations using Virtutech Simics [4] extended with the GEMS toolset [39]. The GARNET [7] network model is used to capture detailed aspects of the interconnection network. We simulate a 36-core CMP system. Each tile consists of an in-order SPARC processor with 32 KB I&D caches. It also includes a 128 KB private L2 cache. We also model 2 memory controllers with DRAM access latency of 90 cycles. The on-chip network is a 6x6 mesh with a detailed router-model that accurately captures our implementation. We explore the design choices for the OMNI network in section 6.1.1, and in section 6.1.2 we present runtime comparisons against other protocols. We run

applications from the SPLASH-2 [3] and PARSEC [11] benchmark suites for our evaluations. SPLASH-2 is a suite of scientific multithreaded applications that has been used in academic evaluations for the past two decades. PARSEC is a more recent benchmark suite that focuses on emerging parallel workloads for shared memory multiprocessors. We run the parallel portion of each workload to completion for each configuration. All benchmarks are warmed up to avoid cold-start effects.

6.1.1 OMNI: Design Choices

Two important choices for the network are the channel width and the number of virtual channels in each virtual network. The channel width impacts the throughput of the network. More importantly, the channel width determines the number of flits in a multi-flit packet, which affects the serialization and eventually the packet latency. The number of VCs also affects the throughput and latency of the network, and consequently the run time of applications. The two parameters can interact with each other and affect the final choice. Figure 6-1 shows the variation in run time as the channel width and number of VCs is varied. We choose a baseline configuration of 16 byte channel width and 4 VCs in each virtual network, unless specified otherwise.

While a larger channel width offers better performance, it also incurs greater overheads – larger buffers, greater link power and larger router area. A channel width of 16 bytes which 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 run time for a few applications. Hence we pick 16 bytes as the channel width. The normalized runtime to a 8 byte channel width network is shown in figure 6-1(a).

Two virtual channels are insufficient for the GO-REQ virtual network which carries the request messages. The request messages are broadcast traffic and require more VCs to handle the load. In addition, we reserve one VC for deadlock avoidance. Hence low VC configurations would degrade the runtime severely. Since there is negligible difference in

runtime between 4 VCs and 6 VCs, we pick 4 VCs since it requires lesser resources in the network. The runtimes shown in figure 6-1(b) are normalized with respect to 4 VCs per virtual network and a channel width of 16 bytes.

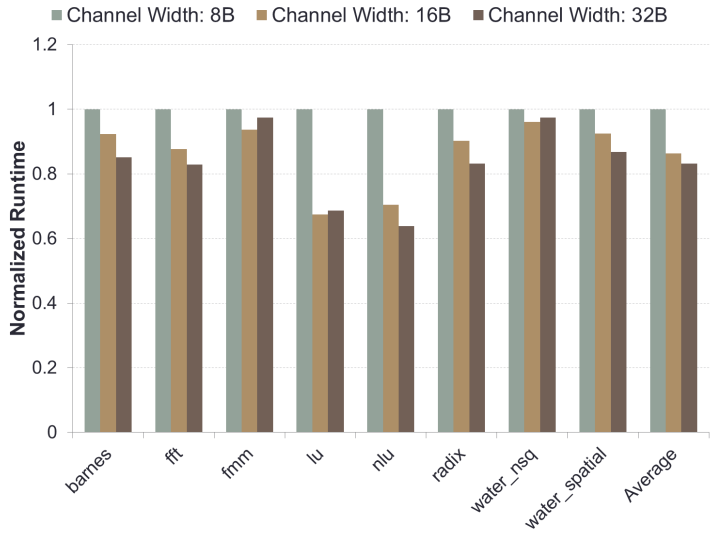
For the UO-RESP virtual network, the number of VCs does not seem to impact the run time greatly once the channel width has been fixed. UO-RESP packets are unicast messages, and generally form a smaller chunk of the packets in comparison to the GO-REQ broadcast requests. Hence 2 VCs is sufficient for this virtual network. Figure 6-1(c) shows the normalized runtime with respect to a channel width of 8 bytes and 2 VCs in the UO-RESP virtual network.

The P2P-REQ virtual network is also expected to have low load, carrying unicast messages to the memory controller. We choose 2 VCs for the P2P-REQ virtual network as well.

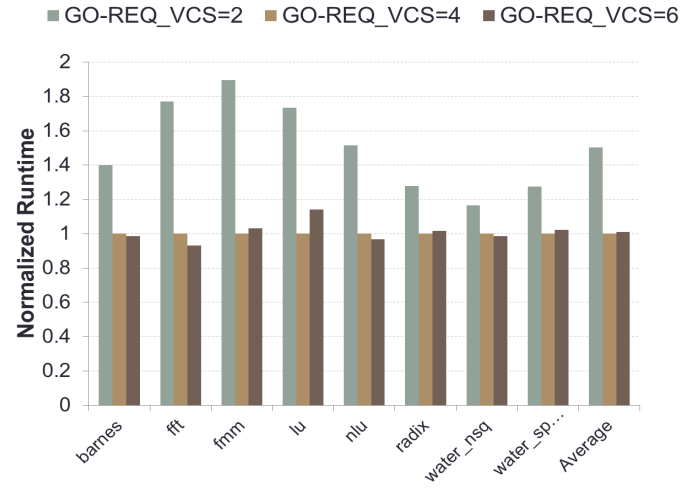
The time window for OMNI was fixed at $2\sqrt{N}+1$. As detailed in section 4.2.2, the smallest time window required for correct execution is $2\sqrt{N}$. Our implementation uses an additional cycle due to latching requirements at the boundary.

Table 6.1: Network Parameters

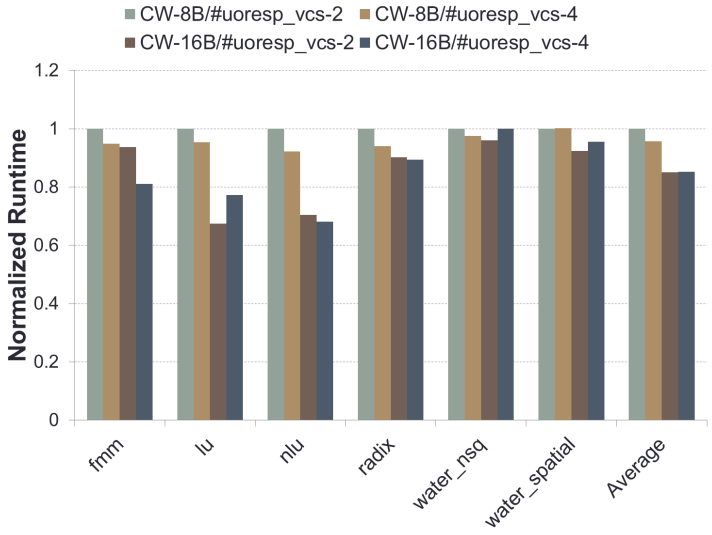
Parameter	Value
Channel Width	16 bytes
Number of GO-REQ VCs	4
Number of P2P-REQ VCs	2
Number of UO-RESP VCs	2
Time Window Size	13 cycles



(a) Channel Width



(b) GO-REQ



(c) UO-RESP

Figure 6-1: Design choices for OMNI

6.1.2 OMNI: Runtime comparisons

We compare OMNI against a baseline directory protocol and the AMD Hammer protocol. The directory protocol stores directory information in the form of a bit-vector to track all sharers for a particular memory block. The directory also acts as a serializing point for all requests. For private L2 configurations, it is difficult to accommodate the entire directory on-chip. Hence we employ an on-chip directory cache, backed up by the memory. Misses in the directory cache result in off-chip memory access with high latency. The Hammer protocol is similar to snoopy protocols in that it broadcasts misses to all nodes. It is designed to allow for implementation on unordered interconnects, by provisioning a home node that serves as an ordering point for misses.

Table 6.2: Simulation Parameters

Parameter	Value
Processors	36 in-order 2-way SPARC cores
L1 Caches	Split I&D, 32 KB 4-way set associative, 1 cycle access time, 32-byte line
L2 Caches	128KB, 4-way, 10-cycle access time, 32-byte line
Memory	2 memory controllers,fully pipelined, 90 cycle DRAM latency + on-chip delay
On-chip network	6x6 2D Mesh, 16-byte links, 2-cycle router pipeline,
Directory Cache Size	256 KB

Table 6.2 summarizes the simulation parameters for the full system evaluations. Figure 6-2 presents the benchmark runtimes for all protocols normalized to the Hammer protocol. OMNI consistently performs better than the directory protocol and Hammer, with an average performance improvement of 36% over both protocols. This benefit is largely due to the avoidance of ordering-point indirection in OMNI.

6.1.3 Filtering Evaluations

There are two primary design choices for incorporating filtering into the network *viz.* region granularity and filter table size. The region granularity presents a tradeoff between

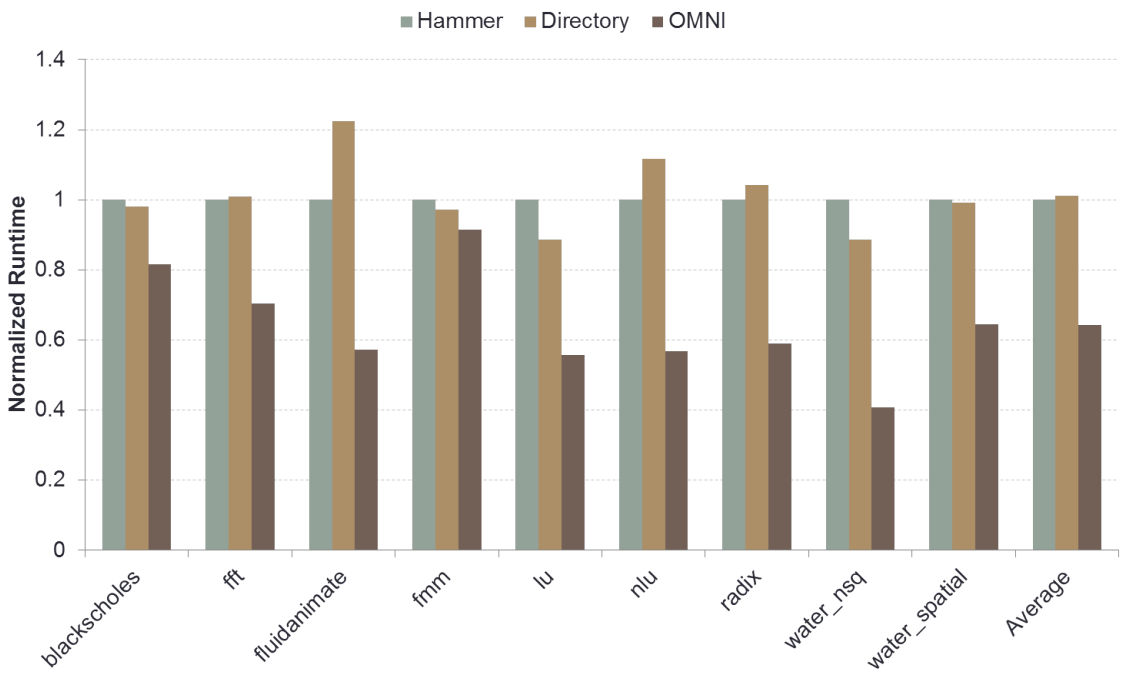


Figure 6-2: Runtime comparison of OMNI compared to Hammer and directory protocols. OMNI consistently performs better, reducing runtime by 36% on average

the number of sharers that can be tracked accurately *vis-à-vis* false sharing, and the area and power consumed by the region tracker. Prior works [8] have shown that a region sizes between 1KB and 4KB represent a good tradeoff between area overhead and sharing information. We pick a region size of 4KB for the SCORPIO processor since it has a lower overhead, and offers good filtering performance. We show architectural evaluations with filter table size of 256 entries.

1. **Ideal filter:** An ideal filter is one that is able to filter away all redundant snoops.

2. **Switch:** We define a *switch* as the process of a flit being routed from an input port to an output port. Thus, a flit that wants to broadcast out of four output ports, has four switches at that router.

3. **Redundant Switch:** A redundant switch is one that eventually leads to a redundant snoop. In other words, every NIC that the flit reaches following this switch,

results in a redundant snoop. Filtering attempts to remove as many redundant switches as possible.

4. **Saved Switch:** A redundant switch that does not need to be performed as a result of filtering is a *saved switch*. The fraction of saved switches is the ratio of the total saved switches to the total number of redundant switches. With ideal filtering information in the network, the fraction of saved switches will be 1. Without network filtering, the fraction of saved switches will be 0.

Since in-network filtering relies on region trackers, we can use the same infrastructure to provide destination filtering. We evaluate the efficacy of destination filtering and in-network filtering below. Figure 6-3 shows the efficiency of the destination filter and the network filters, as compared to an ideal filter. The efficiency of the destination filter is directly related to the efficiency of the region tracker, and therefore affected by the region size and number of entries. The efficiency of the network filters is bounded above by the efficiency of the region tracker, and hence destination filtering. Figure 6-3 shows that the destination filter's efficiency is about 50% as compared to an ideal filter. The network filters achieve about 40% filtering efficiency as compared to the ideal filter, which is also quite close to the destination filter's efficiency of 50%. However, the bottom graph in figure 6-3 shows that the percentages of switching saved is only around 25%. This suggests that the filtering of the flits happens very close to the destinations. In other words, the propagation of filtering information is not very efficient. We analyze possible reasons for this in section 6.2.3.

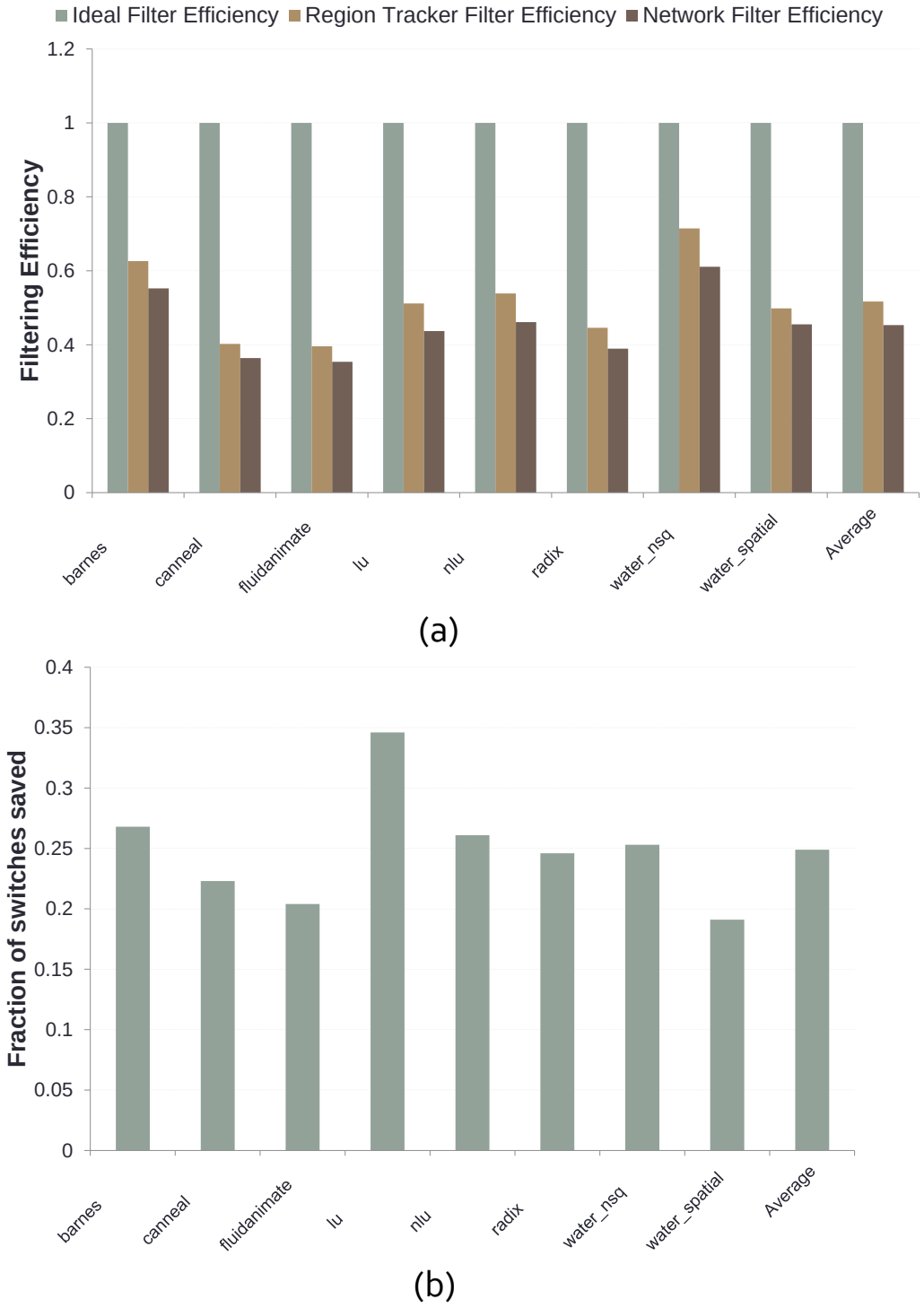


Figure 6-3: (a) Filtering Efficiency (b) Fraction of switches saved

6.2 Network Evaluations

The OMNI network was implemented using *System Verilog*. The implementation is parameterized offering flexibility in terms of number of nodes, channel width, number of virtual networks, number of VCs, number of flits per packet and number of buffers. Each of these parameters may be varied independently through parameters in a configuration file. The final parameters for OMNI were set according to table 6.1.

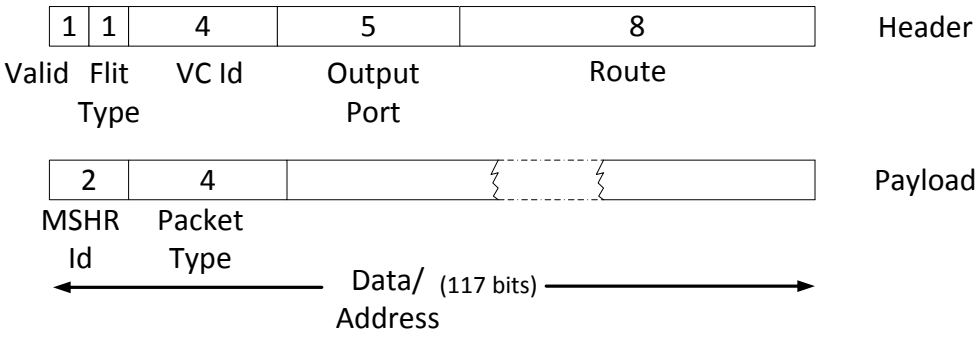


Figure 6-4: Structure of flit (numerical values in bits)

The network RTL was verified at different levels. We perform unit tests on the network by injecting various synthetic traffic patterns – uniform random (all nodes equally likely to send and receive), permutation (send to single destination), neighbor (send packets to neighbors) and broadcast traffic. We verify that all packets are received, and for the GO-REQ virtual network, that all packets are received at all nodes in the same global order. Following this, the network RTL was integrated with the rest of the SCORPIO RTL design. We verified the functionality of the complete system at the RT-level, using a regression test suite that exerts multiple aspects of the system design. A brief summary of the regression tests is shown in table 6.3 – many of the tests rely on the correct functionality of the network, including the ordering guarantees provided by OMNI. We also implement hardware testers for the network, that allows us to inject and collect statistics on traffic in the fabricated SCORPIO chip.

Table 6.3: Regression test suite: These represent the broad categories of regression tests that were used to verify the functionality of SCORPIO and OMNI

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	Loads data from the flash memory and stores them 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.

6.2.1 Performance

We evaluate the network RTL to determine the latency and throughput characteristics of the implemented network. Figure 6-5 shows the latency statistics for the UO-RESP network. We inject 1 flit UO-RESP packets in uniform random traffic to obtain the latency curves. The low load latency is around 10 cycles, and the throughput of the network improves as we increase the number of VCs. Since the target applications for the SCORPIO chip (PARSEC, SPLASH-2) have low injection rates, and are unlikely to stress the network, 2 VCs is sufficient for the UO-RESP network as shown in section 6.1.1. The flattening of the latency curves is because of finite buffering at the network interface controllers.

Figure 6-6 shows the latency curves for the P2P-REQ virtual network when injected with uniform random traffic. Increasing the number of VCs has reduced benefit here. This is because of the point-to-point ordering requirement on this virtual network. Our implementation prioritizes the earlier flit to maintain point-to-point ordering in the P2P-REQ virtual network. Thus later flits are blocked until earlier ones leave, even if they have the necessary resources to proceed to the next router.

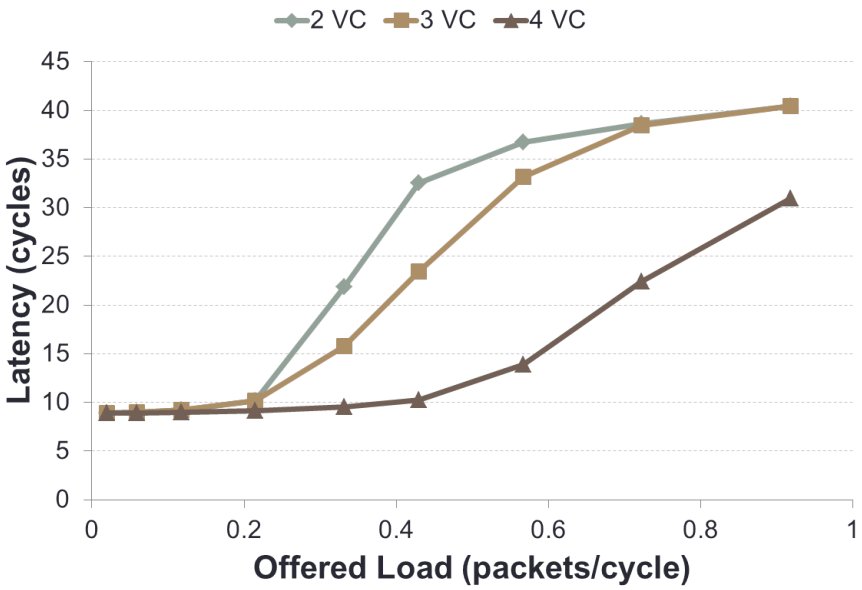


Figure 6-5: Network Performance for UORESP packets

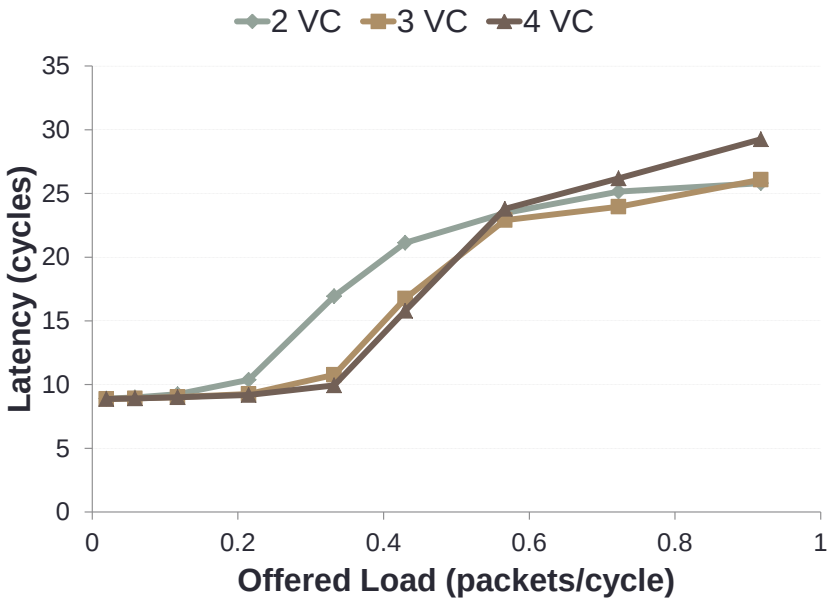


Figure 6-6: Network Performance for P2P-REQ packets

The GO-REQ virtual network saturates quickly due to the broadcast nature of the traffic on the virtual network. We can derive an upper bound on the throughput for broadcast traffic as follows. We assume that all nodes inject packets at a constant injec-

tion rate, and each node is equally likely to inject a packet in a given cycle. Because of the global ordering requirement, the maximum rate at which a destination node can service packets from a particular source node is $1/N$ - here we assume that on an average each node receives a notification from every node every time window, and hence we have to cycle through each node in the global order. This places an upper bound on the throughput as $1/N = 0.027$.

The low load latency of packets in the GO-REQ network was found to be around 30 cycles. We provide a rationale for this value. Suppose that a packet is injected at $t = 0$. The average number of hops in a $\sqrt{N} \times \sqrt{N}$ mesh [18] for a packet is $\sqrt{N}/2 = 3$. We add two cycles for the transfers into the network interfaces at the source and destination nodes, which brings the number upto 5 cycles. However a packet delivered to its destination may not be processed immediately. The destination node must receive the notification and determine the global order for the packet. Once a packet has been injected, its notification may be injected only at the start of the next time window. This notification is then processed by all nodes at the end of the time window, and consequently the order for the packet is fixed on average after $T_{win} + T_{win}/2 \approx 19$ cycles, where T_{win} is the length of the time window. We add $T_{win}/2$ since the packet itself may be injected at any point within the time window. On average, we may consider the packet to be injected in the middle of the time window. Therefore, regardless of when the packet is received, the earliest it may be taken up for processing is at $max\{5 \text{ cycles}, 19 \text{ cycles}\} = 19$ cycles. In addition to this latency, the packet may be globally ordered behind packets from other nodes. If we assume all nodes are injecting packets, then on average the packet may have to wait an additional 18 cycles (we have 36 nodes in the system). Therefore, the latency of the packet could be around 37 cycles. However, at low load not all nodes are likely to be injecting at the same time. Hence we might observe smaller latencies. This is only a rudimentary sketch, that seeks to highlight the possible contributors to packet latency. Since the notification delay is a constant cost in OMNI, an interesting analysis would be to determine when this delay becomes prohibitive in comparison to ordering point

indirection as in the case of a directory node. It would also be of interest to study the variance in the waiting time for packets once they are delivered to the destination nodes. We differ these analysis to future work.

6.2.2 Operating Frequency, Area and Power

The OMNI router and network interface were synthesized, and placed-and-routed in a commercial 45 nm technology. We also synthesized the network with filtering enabled to evaluate the cost of adding filtering. Figure 6-7 shows the critical paths in the network. The process of generating the SA-O request, performing the arbitration for the crossbar and generating the grants for the input ports occupies a majority of the critical path in both cases. Addition of filtering leads to a longer critical path due to the added delay of updating and reading the filter table. The OMNI router and network interface were synthesized, at 1 GHz, while the addition of filtering causes the operating frequency to drop to 900 MHz.

The network interface and router together occupy an area of $0.16mm^2$. Figure 6-8 shows the post-synthesis component wise breakdown of area for the network interface and router. The buffers consume around 60% of the area. We also observe that the overhead of the notification network is minimal (around 1%). With the addition of filtering, the total area goes up by 20%, a majority of which is the area overhead of the filter table.

We evaluated the post-synthesis power of the network using Synopsys PrimePower. We inject uniform random traffic through the network and pass the corresponding activity traces to PrimePower. The average power of the network was between 80mW and 100mW for various injection rates. Figure 6-9 shows the power breakdown for the network. We observe that the buffer and VC state (in the router and NIC) combined account for close to 50% of the total network power. We also note that the VC state power is quite comparable to the buffer power in the input ports of the router. This is because the buffers in the network are clock-gated, while the registers in the VC-state are not. Non-gated registers consume clocking power even when idle, leading to higher power.

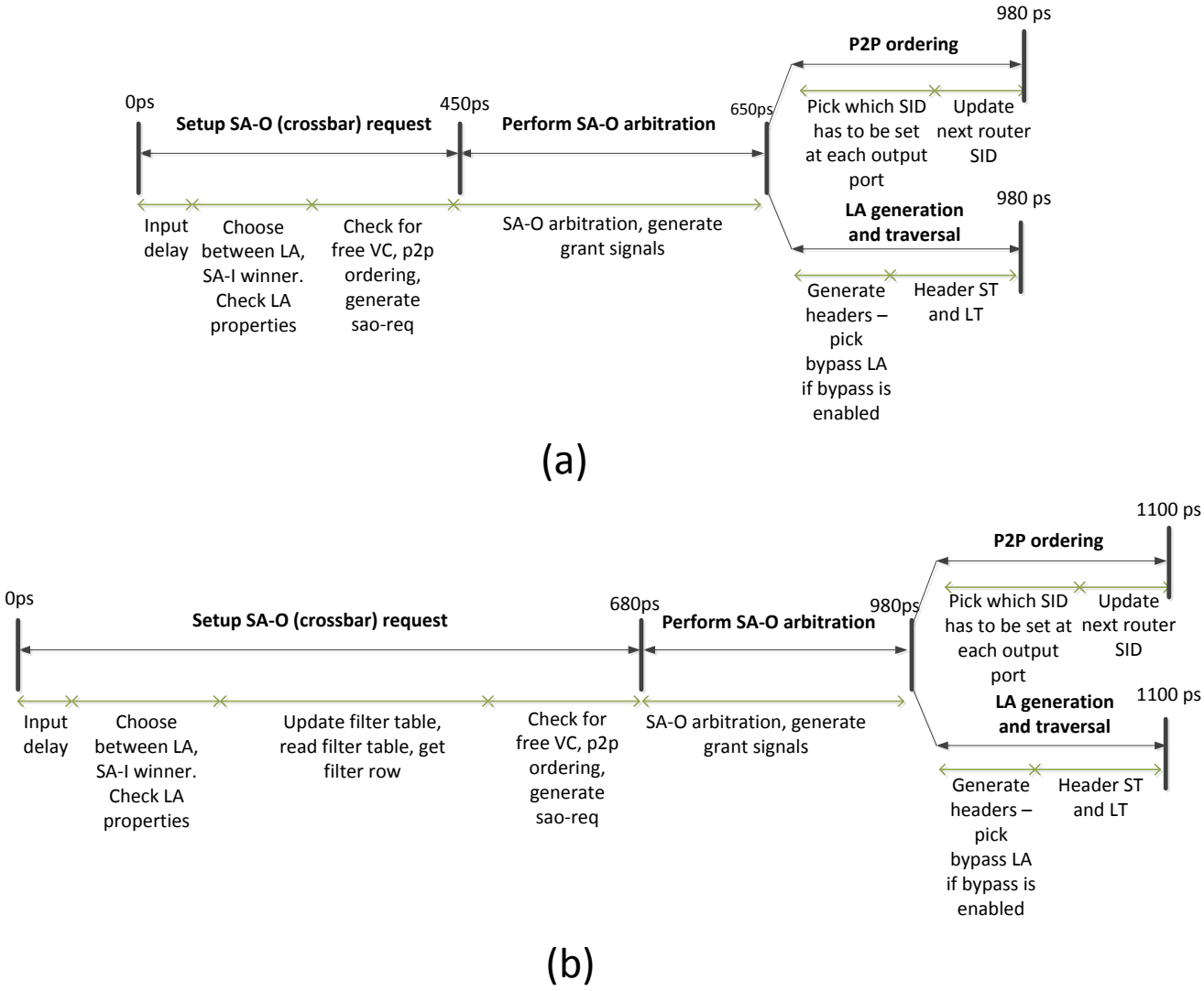


Figure 6-7: Post-synthesis critical paths for (a) OMNI (b) OMNI with filtering

6.2.3 Exclusion of filtering from SCORPIO prototype

In section 6.1.3 we observed that while network filtering is successful in filtering away redundant snoops, most of the filtering happens close to the destinations. One reason for this could be the inefficient propagation of filtering information in the network. On receiving a non-sharing request, the router queries the filter table to determine if the request must be propagated to its neighbors as per the pseudo-code in chapter 5. Now if the LOCAL port is not set, or if there is no entry corresponding to the region in the filter

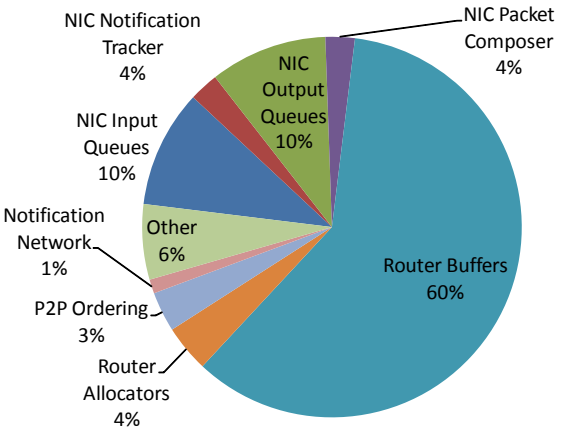


Figure 6-8: Breakdown of area for router and network interface

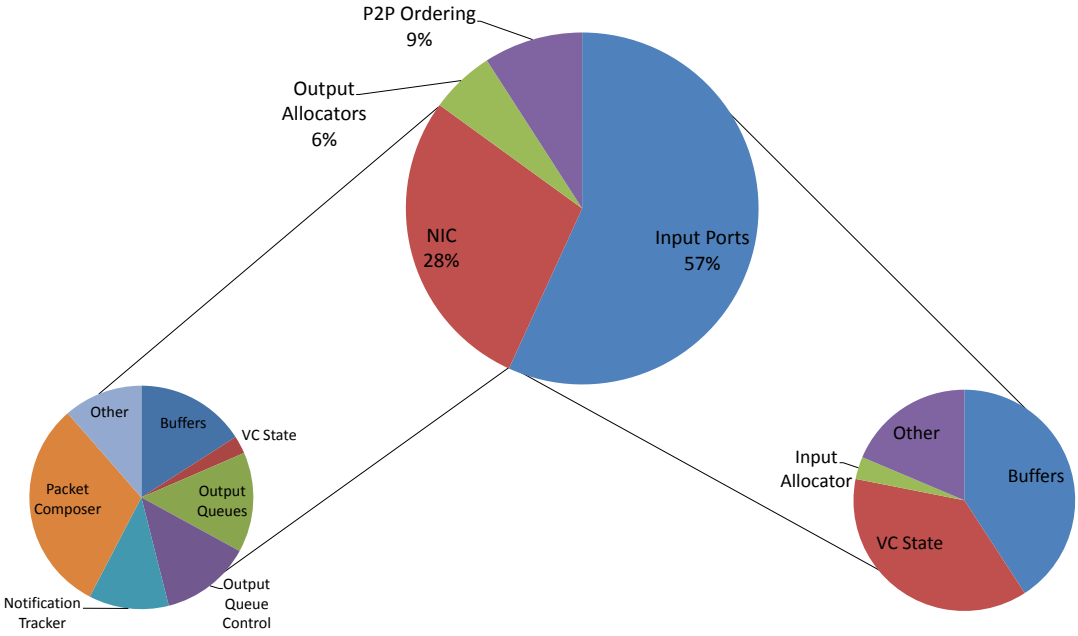


Figure 6-9: Power breakdown for network components

table, then the non-sharing request is not propagated. Unless the region tracker populates the router filter table appropriately by injecting non-sharing updates, it may lead to non-sharing updates being dropped by intermediate routers. This restricts the propagation of filtering information, and consequently the effectiveness of network filtering.

The SCORPIO chip places memory controllers on the north and south boundaries of the chip. The memory interface controller is connected to cores 3, 4, 33 and 34. Because

of the distribution of the addresses across the different memory controllers, it is difficult to implement a region tracker for the memory controller. Therefore, we do not receive any sharing or non-sharing updates from the memory controllers. Consequently, there can be no filtering on any path leading up to the memory controller.

The addition of filtering also leads to a decrease in the operating frequency. In addition, a majority of the power consumption in our system is due to clock power. Further, the network power is only a small fraction of the chip power budget (figure 6-10. For these reasons, we felt filtering might have a negligible impact on the power and performance of the system, and hence it was decided not to include the same in the SCORPIO prototype.

6.3 SCORPIO Chip

Figure 6-10 shows the post-layout snapshot of the entire SCORPIO chip, with the tiles and other components annotated. The OMNI router and network interface occupy around 10% of the area of the tile. The entire chip was placed and routed with an operating frequency of 833 MHz.

6.4 Chapter Summary

We laid out the design choices for OMNI and the rationale behind the same. We then evaluated our in-network global ordering scheme as implemented in OMNI, and compared the performance to other coherence protocol implementations. OMNI consistently provides a performance benefit, with an average reduction of 36% in the runtime of SPLASH-2 and PARSEC benchmarks. The OMNI network was implemented in SystemVerilog, integrated with the rest of the SCORPIO design and functionally verified. The post-synthesis area and power estimates show that the network occupies 10% of the area and consumes around 14% of the power.

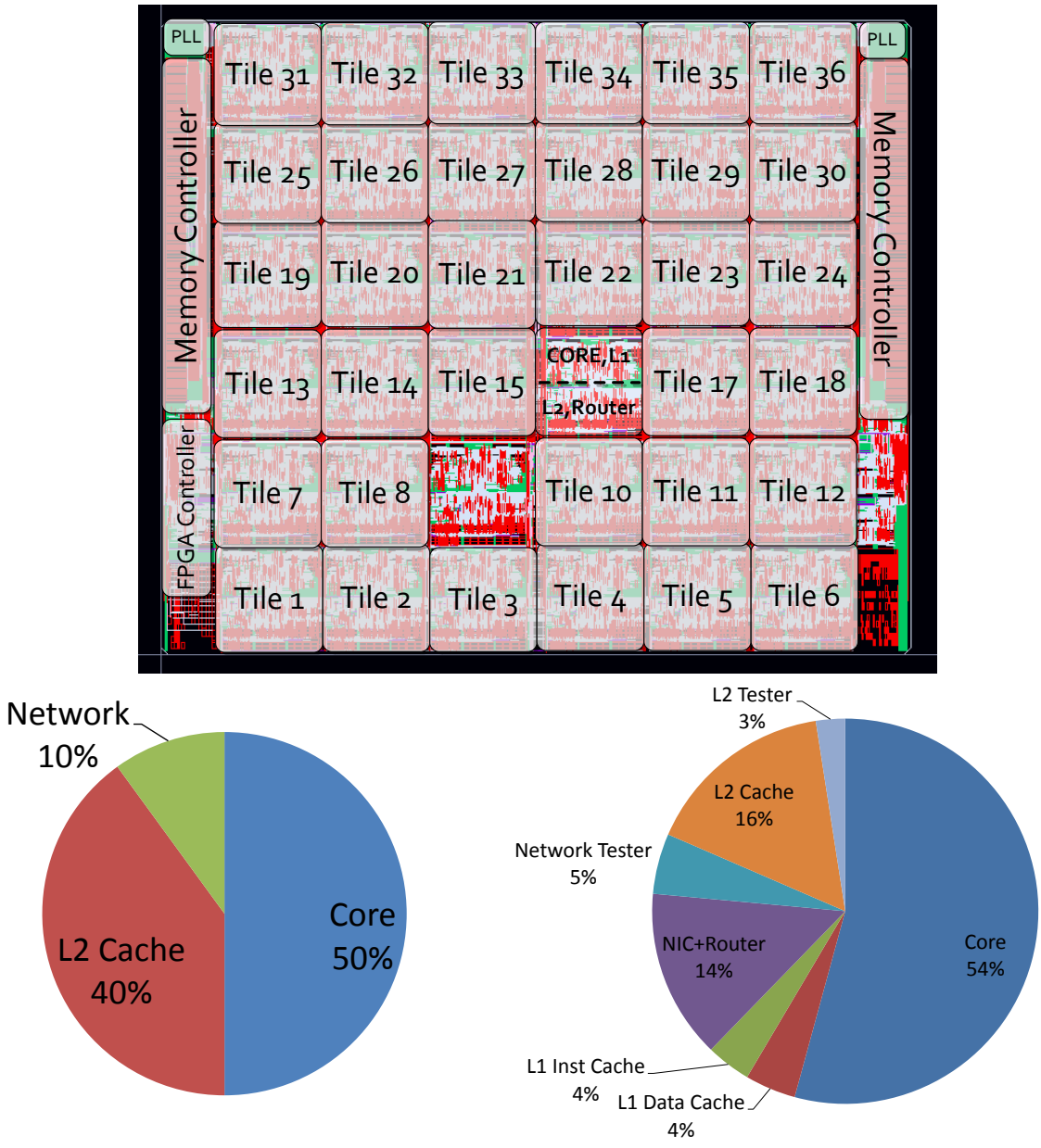


Figure 6-10: Annotated layout of SCORPIO chip (top) and breakdown of area and power of a tile (bottom)

7 Conclusions

The many-core CMP era presents a host of challenges to computer architects. Specifically, scalable cache coherence is an important problem that needs to be addressed, in order to sustain the performance growth predicted by Moore's Law, while providing a shared memory abstraction. At the same time, packet-switched NoCs are rapidly becoming a key enabling technology for CMPs. The increasing interaction between coherence mechanisms and communication substrates is an important consideration for future multicore designs. This thesis has been aimed at leveraging the interconnection network to enable scalable coherence for many-core CMPs. Specifically, it focused on realizable mechanisms for in-network coherence.

7.1 Thesis Summary

We proposed a distributed in-network global ordering scheme that enables snoopy coherence on unordered interconnects. We presented details on OMNI (Ordered Mesh Network Interconnect), a realization of the proposed ordering scheme on a two-dimensional mesh network-on-chip. We elucidated microarchitectural details of OMNI, highlighting subtle implementation issues and techniques for handling the same. We also explored mechanisms for enabling in-network filtering techniques in OMNI. OMNI-enabled snoopy coherence was shown to be effective, providing 36% improvement in runtime in compar-

ison to directory and Hammer protocols.

OMNI is an integral component of the SCORPIO processor – a 36-core multicore processor prototype, supporting snoopy coherence, and implemented in a commercial 45nm technology. OMNI occupies around 10% of the area of the chip, while consuming less than 100mW of power. The SCORPIO chip, including OMNI, was laid out at a frequency of 833 MHz.

7.2 Future Work

The in-network global ordering scheme and the notification architecture proposed in this thesis provide a mechanism to decouple message delivery from ordering. This also opens up avenues for other architectures for the notification network, and other architectures to enable distributed ordering in on-chip interconnection networks. The proposed lightweight notification network, could find other utility, especially in transactions that require a tight latency bound.

Another avenue for exploration is more efficient propagation of filtering information in the network. An associated question is the efficacy of in-network filtering in the context of efficient multicast support in the network routers.

We also await the return of the SCORPIO chip, to perform measurements, and to correlate our simulation and pre-fabrication results against real silicon.

Bibliography

[1] An Introduction to the Intel Quick Path Interconnect. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

[2] Intel Single Chip Cloud Computer. http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html.

[3] SPLASH-2 benchmark suite. http://www-flash.stanford.edu/apps/SPLASH.

[4] Virtutech AB. Simics full system simulator. http://www.virtutech.com/.

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

[6] Anant Agarwal et al. An evaluation of directory schemes for cache coherence. In *Computer Architecture News*, May 1988.

[7] Niket Agarwal et al. GARNET: A detailed on-chip network model inside a full system simulator. In *ISPASS*, April 2009.

[8] Niket Agarwal et al. In-Network Coherence Filtering: Snoopy Coherence without broadcasts. In *IEEE Symposium on Microarchitecture (MICRO)*, December 2009.

[9] Niket Agarwal et al. In-network snoop ordering: Snoopy coherence on unordered interconnects. In *International Symposium on High Performance Computer Architecture (HPCA)*, February 2009.

[10] Luiz Andre Barroso et al. Memory system charecterization of commerical workloads. In *International Symposium on Computer Architecture (ISCA)*, June 1998.

[11] Christian Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2008.

[12] Ender Bilir et al. Multicast snooping: A new coherence method using multicast address network. In *International Symposium on Computer Architecture (ISCA)*, June 1999.

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

[14] Alan Charlesworth. Starfire: Extending the SMP envelope. In *IEEE Symposium on Microarchitecture (MICRO)*, November 1998.

[15] Thomas Chen et al. Cell Broadband Engine architecture and its first implementation. In *IBM Developer Works*, November 2005.

[16] Pat Conway and Bill Hughes. The AMD Opteron Northbridge Architecture. volume 27, March-April 2007.

[17] 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.

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

[19] Robert H. Dennard et al. Design of Ion-Implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid State Circuits (JSSC)*, 9(5):256–268, 1974.

[20] Noel Eisley et al. In-Network cache coherence. In *International Symposium on Computer Architecture (ISCA)*, December 2006.

[21] Noel Eisley et al. Leveraging on-chip networks for data cache migration in chip multiprocessors. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2008.

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

[23] James R. Goodman. Source snooping cache coherence protocols. `http://parlab.eecs.berkeley.edu/sites/all/parlab/files/20091029-goodman-ssccp.pdf`.

[24] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *International Symposium on Computer Architecture (ISCA)*, June 1983.

[25] Paul Gratz et al. Implementation and Evaluation of On-Chip Network Architectures. In *International Conference on Computer Design (ICCD)*, 2006.

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

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

[28] Yatin Hoskote et al. A 5-GHz mesh interconnect for a Teraflops processor. *IEEE Symposium on Microarchitecture (MICRO)*, September 2007.

[29] David J.Schanin et al. The design and development of a very high speed system bus-the Encore Multimax Nanobus. In *Proceedings of ACM Fall Joint Conference*, November 1986.

[30] Michael Kistler et al. Cellmultiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3), May-June 2006.

[31] Avinash Kodi et al. Design of energy efficient channel buffers with router bypassing for network-on-chips (NoCs). In *ISQED*, March 2009.

[32] Amit Kumar et al. Express virtual channels: Toward the ideal interconnection fabric. In *International Symposium on Computer Architecture (ISCA)*, June 2007.

[33] Amit Kumar et al. Token flow control. In *IEEE Symposium on Microarchitecture (MICRO)*, November 2008.

[34] Jeffrey Kuskin et al. The Stanford FLASH multiprocessor. In *International Symposium on Computer Architecture (ISCA)*, April 1994.

[35] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, (9):690–691, 2005.

[36] Daniel Lenoski et al. The directory-based cache coherence protocol for the DASH multiprocessor. In *International Symposium on Computer Architecture (ISCA)*, April 1990.

[37] ARM Ltd. Advanced microcontroller bus architecture (amba) specifications. `http://www.arm.com/products/system-ip/amba/amba-open-specifications.php`.

[38] Milo Martin et al. Timestamp snooping: An approach for extending SMPs. In *ASPLOS*, December 2000.

[39] Milo Martin et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) toolset. In *SIGARCH Computer Architecture News*, 2005.

[40] Michael R. Marty and Mark D. Hill. Coherence Ordering for Ring-based chip multiprocessors. In *IEEE Symposium on Microarchitecture (MICRO)*, December 2006.

[41] Andreas Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *International Symposium on Computer Architecture (ISCA)*, May 2005.

[42] Andreas Moshovos et al. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2001.

[43] Li-Shiuan Peh and Natalie E. Jerger. *On-Chip Networks*. Morgan and Claypool, 2009.

[44] Valentina Salapura et al. Design and implementation of the Blue Gene/P snoop filter. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[45] Karthikeyan Sankaralingam et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *International Symposium on Computer Architecture (ISCA)*, June 2003.

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

[47] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool, 2011.

[48] Karin Strauss et al. Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors. In *IEEE Symposium on Microarchitecture (MICRO)*, 2007.

[49] Michael Bedford Taylor et al. Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams. In *International Symposium on Computer Architecture (ISCA)*, June 2004.

[50] David Wentzlaff et al. On-chip interconnection architecture of the Tile processor. *IEEE Symposium on Microarchitecture (MICRO)*, 27(5):15–31, September 2007.

[51] Craig Williams et al. Delta coherence protocols. *IEEE Concurrency*, 8(3), July 2000.

[52] Ling Xin and Chiu-Sing Sing Choy. A low-latency NoC router with lookahead bypass. In *ISCAS*, May 2010.

[53] Jason Zebchuk et al. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. In *IEEE Symposium on Microarchitecture (MICRO)*, December 2007.