Design and Evaluation of Network Interfaces for System Area Networks

by

Shubhendu Sekhar Mukherjee

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1998

Abstract

Much of a computer's communication performance is determined by how well it interacts with networks. Such interaction is critical for latency-sensitive applications, such as parallel programs that send frequent, short messages. Fortunately, networks have improved dramatically, especially System Area Networks (SANs). SANs provide sub-microsecond latency, gigabytes per second bandwidth, and very high reliability to 10-100 hosts. Unfortunately, this dramatic improvement in network performance is seldom delivered to applications. A key bottleneck is the host network interface (NI), which connects a network to a host computer. For example, conventional NIs are usually accessed via direct memory access or uncached, memory-mapped device registers, which can incur latencies between ten and hundreds of microseconds.

This thesis investigates novel techniques to improve interactions between a processor and a SAN NI. A key principle underlies these techniques: *treat NI access as regular, side-effect-free memory access, and not as a disk interface access*. The thesis' first contribution shows that such treatment opens up at least eight opportunities for improving processor-NI interactions.

This thesis' second contribution is the design and evaluation of a novel class of NIs called *Coherent Network Interfaces* (CNIs). CNIs realize the key principle enunciated in this thesis. CNIs appear to their hosts more like regular, cachable memory than like a disk interface and exploit all eight opportunities for improving processor-NI interactions.

The thesis' third contribution is a systematic classification and evaluation of NI data transfer and buffering parameters. I evaluate these parameters in the context of several commercial and research NIs.

Finally, this thesis' last contribution is the *Cosmos* coherence message predictor. This part differs significantly from the rest of this thesis. Coherence protocol message prediction can accelerate the performance of directory protocols, which are used by most large shared-memory multiprocessors to keep per-processor caches coherent. Cosmos predicts the source and type of the next coherence message for a cache block using general prediction logic that is extension of Yeh and Patt's PAp branch predictor. For five scientific applications running on 16 processors, Cosmos' prediction accuracy ranges between 62% and 93%.

Acknowledgments

Many people helped in making this dissertation possible and my bumpy ride through graduate school bearable. In particular, I am greatly indebted to two people: my advisor Mark Hill and my wife Mimi. Mark provided valuable guidance, encouragement, and criticism from the very first day I came to Wisconsin. Mark likes to teach by examples. To me Mark himself was the example and a role model. By watching Mark I learned how to discipline and balance my professional and social lives. I am thankful to him for the patience and the understanding with which he taught me the tricks of our trade.

In every step Mimi supported and encouraged me to complete my dissertation. She not only celebrated my successes, but also endured all my frustrations. Without her constant companionship and understanding I would not have finished this dissertation.

Jim Larus, Guri Sohi, and David Wood were great inspirations. Jim and David were a constant source of novel ideas. Guri's insightful comments helped improve my work.

My family helped and encouraged me in many different ways. My father, who did a Ph.D. himself, always encouraged me to pursue a Ph.D. My mother, who founded her own school, taught me that you can achieve anything in life if you set your mind to it. My brother (Dipu), sister-in-law (Tutu), and Mimi's sister (Riki) helped provide a supportive family environment. Mimi's parents' provided advice and support whenever needed. Finally, both my and Mimi's family patiently endured my absence from many family reunions.

The Wisconsin Wind Tunnel project, colleagues outside Wisconsin, and the Computer Sciences department at UW-Madison provided me with tools I needed to complete this dissertation. In particular, Steve Reinhardt, Babak Falsafi, and Mike Litzkow helped develop and debug the Wisconsin Wind Tunnel II (WWT II) parallel simulator, which I used for most of my experiments in this thesis. Fred Chong and Shamik Sharma provided me with one of the benchmarks (*spsolve*) I used in this thesis. The Computer Sciences department provided me with a tremendous amount of computing resources I needed for my simulations. In particular, the seven Sun Enterprise servers—most of them equipped with 16 250-MHz UltraSPARC processors—and the 40-node Wisconsin COW (Cluster of Workstations) provided rapid turnaround time for my parallel simulations with WWT II.

The architecture faculty at Wisconsin—Mark Hill, Jim Goodman, Jim Smith, Guri Sohi, and David Wood—have created a congenial atmosphere in which graduate students can interact freely and openly with faculty. The architecture students at Wisconsin were always willing to listen to my ideas and provide valuable criticism about them. In particular, Vijay—presently Prof. Vijaykumar at Purdue University—patiently listened to and criticized many of my ideas and provided valuable advice regarding the mechanics of graduate school.

Several people provided valuable feedback and information for this thesis. Mark Hill, Guri Sohi, and David Wood patiently read through this thesis. Nick Carter, Bob Felderman, Mike Galles, Leonidas Konthothanassis, Whay Lee, Ken Mackenzie, Toshi Shimizu, and Bob Zak provided valuable information regarding several commercial and research networks and network interfaces. Tom Anderson, Babak Falsafi, Suresh Chalasani, Satish Chandra, Erik Hagersten, Rebecca Hoffman, Stefanos Kaxiras, Larry Landweber, Jim Larus, Ken Mackenzie, Rich Martin, Subbarao Palacharla, Larry Peterson, Steve Reinhardt, Avinash Sodani, T.N.Vijaykumar, Jon Wade, and Bob Zak provided valuable comments on this work.

My dissertation research was supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, National Science Foundation with grants MIP-9225097, MIPS-9625558, and CDA-9623632, and donations from Sun Microsystems.

Finally, I would like to thank *Eureka Joe's* for providing me with a constant supply of cappucino and a place to sit, think, and relax during my tenure in graduate school.

Table of Contents

List of Figures

List of Tables

For your dissertation you must think of one thought that no one has thought before.

— Pasupati Mukerjee, Emeritus Professor,

School of Pharmacy, U. of Wisconsin-Madison

Chapter 1

Introduction

The term “computer” is perhaps a misnomer today. A modern computer system often “communicates” with a communication network more than it “computes.” Consequently, much of a computer’s value depends on how well it interacts with networks. To enhance this value, designers must improve the communication performance delivered to users. The aspect of communication performance that is quoted most commonly is *bandwidth*. Bandwidth is the rate at which data can flow through the network and computer. High bandwidth is critical when transmitting high-quality video or large files. An under-appreciated aspect of communication is *latency*. Latency is the user-to-user delay for sending a message. Latency determines performance of applications that send many small messages, as can be found in fine-grain parallel computing, network file systems, database lock managers, and world-wide web requests.

Latency of communication can be broken into three important components (Figure 1-1):

- latency through software protocols that generate and consume messages,
- latency through the network, and

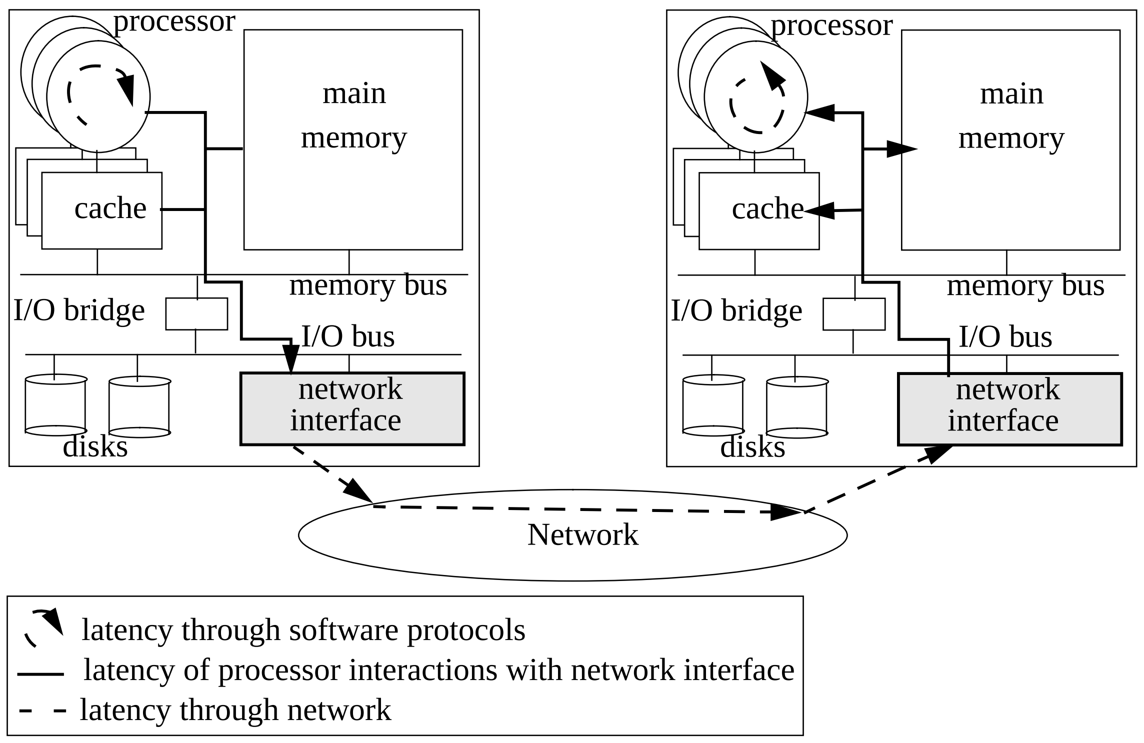


Figure 1-1. Three components of message latency.

- latency of processor interactions with a *network interface (NI)* that connects a computer with a network.

The advent of high-performance microprocessors with supercomputer-like clocks, lean software protocols (e.g., [127, 128]), and high-speed reliable networks with tens of nanoseconds latency (e.g., [41]) have drastically reduced the impact of software protocols and networks on the overall latency of communication. Consequently, the third component—processor interactions with an NI—threatens to become a critical bottleneck, particularly for a cluster of workstations connected with a high-speed network.

An NI is a device that sends and receives messages from the network on behalf of the computer. To send or receive a message a processor (and/or memory system) must interact with the NI by reading and writing messages into the device. Conventional NIs were designed for slower processors, heavy-weight protocols, and slow and unreliable networks. Consequently, processor interactions with conventional NIs incur large components of latency. For example, conventional NIs are usually accessed via low-level software (e.g., device driver) inside the operating system, located on slower I/O buses, and accessed via direct memory access (DMA) or uncached, memory-mapped device registers. Each of the these components can add between ten and hundred microseconds latency to the total latency seen by a message [5, 127, 11].

The key problem with conventional NIs is that a processor interacts with them in almost the same way it interacts with disk interfaces. For example, disk-resident files are usually accessed via the operating system and read into memory using direct memory access (DMA). However, technological advances have made current networks between four to five orders of magnitude faster than disks. Consequently, NIs must be redesigned to accommodate this new generation of high-speed networks.

This thesis proposes and evaluates novel techniques to improve a processor and memory system's interactions with an NI. That is, this thesis examines techniques to improve an NI's internal interface to a computer and *not* its external interface to networks. These techniques reduce the overall message latency to a few microseconds. Other related projects and designs have proposed using reflective memory techniques to directly deposit data from a processor's cache or memory in one node to another node's memory [44, 12]. The techniques described in this thesis can further improve the performance of communication via reflective memory. This is because the techniques proposed in this thesis improves the important component of latency: processor-NI interactions, which is critical even for reflective memory (see Chapter 5).

The NI techniques I propose in this thesis are most relevant to a Cluster of Workstations (COW). A COW is a parallel machine built with commodity workstations and connected with a high-speed and reliable System Area Network (Appendix A). COWs are attractive over traditional massively parallel machines (MPPs) because COWs have the potential to offer performance similar to MPPs, but at a reduced price. The reduced price of COWs arise from the commodity nature of its parts—commodity workstations, commodity operating system, and (perhaps) commodity network.

This thesis makes four contributions:

- The first contribution of this thesis (Chapter 2) is the development of a key NI design principle: *treat NI access as regular, side-effect-free memory access, and not as a disk interface access*. I show how treating a processor's access to an NI as a regular memory access can significantly improve processor-NI interactions.
- The second contribution of this thesis is the design (Chapter 3) and evaluation (Chapter 4) of a novel class of NIs called *Coherent Network Interfaces (CNIs)*, which interact with a processor and memory system via cachable, coherent memory operations. CNIs are the embodiment of the design principle mentioned above.
- The third contribution of this thesis (Chapter 5) is a systematic classification, examination, and evaluation of NI data transfer and buffering parameters, which have significant impact on the performance of processor-NI interactions.
- Finally, the fourth contribution (Chapter 6) differs significantly from the rest of this thesis. Techniques discussed in the rest of the thesis can accelerate user-to-user messaging in a parallel machine programmed with a message-passing programming model. In contrast, Chapter 6 examines techniques to accelerate the communication performance of parallel machines that are programmed with a shared-memory model. Most large shared-memory machines use a directory protocol to keep per-processor caches coherent. Unfortunately, these protocols often incur long latencies due to either long protocol actions or multiple message exchange. This chapter proposes and evaluates the *Cosmos* coherence protocol message predictor, which can help ameliorate this latency by predicting with high accuracy the next incoming coherence message for a cache block.

Finally, Chapter 7 summarizes this thesis and provides directions for future work. Appendix A describes the characteristics of a new generation of networks called system area networks.

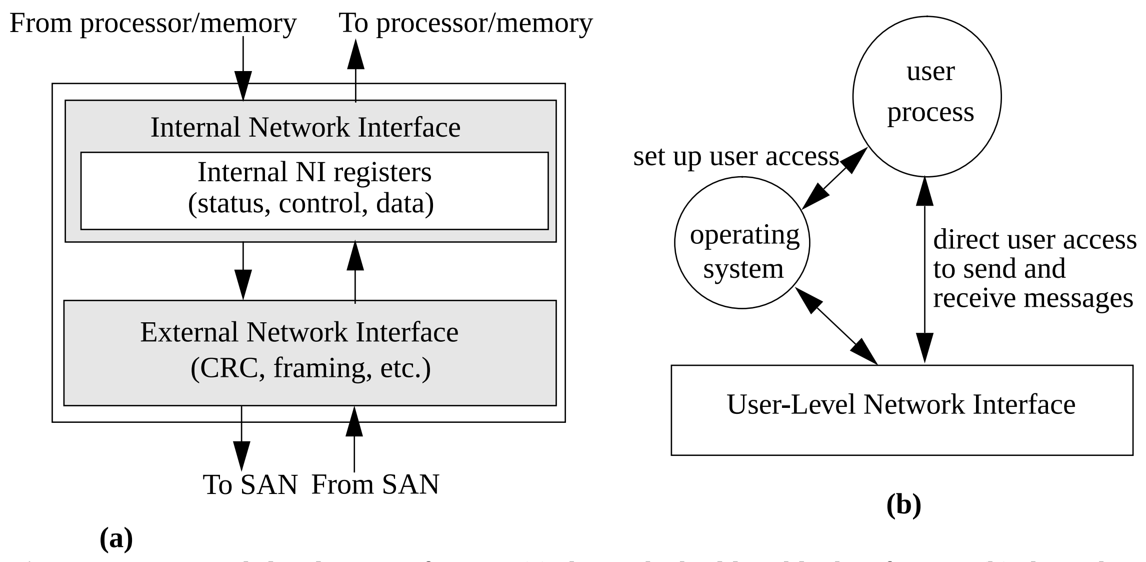


Figure 1-2. High-level views of an NI. (a) shows the building blocks of an NI. (b) shows how a user process can logically access an NI.

The rest of this chapter is organized as follows. Section 1.1 describes the key components of an NI. Section 1.2 outlines problems with conventional NI designs. Section 1.3 argues why these problems will become even more critical in the future. Section 1.4 states my proposed solutions and discusses contributions of this thesis. Section 1.5 motivates and examines the Cosmos coherence message predictor.

1.1 Components of a Network Interface

A network interface (NI) in a host node is a device that allows a processor to send and receive messages to and from a network that connects these host nodes. The network accepts messages from an NI and delivers them to one or more NIs connected to the network. An NI consists of two parts, the internal NI and the external NI (Figure 1-2a). I define the internal NI as the NI's interface to the processor, main memory, and (perhaps) disks, and external NI as the NI's interface to the network. The internal NI contains logic and memory that the processor uses to send and receive messages to and from the NI. For example, a processor can send a message to the network by writing messages to the data registers of the internal NI In contrast, an external NI performs network-specific functions, such as cyclic-redundancy checks, network-specific framing, etc.

This thesis examines the architecture of an internal NI. An internal NI consists of two parts: the send interface and the receive interface. Each interface consists of four components: status registers, control registers, data registers, and an optional notification mechanism. *Here I use the term registers just as an architectural specification*; the registers, for example, may be implemented with DRAM. I will examine alternative implementations of these registers and notification mechanism later. In this section, I discuss the function of each of the components.

Status Registers. NI status registers contain NI device status information. A receive interface status register, for example, can indicate that a new message has arrived from the network, and a send interface status register can indicate that the NI has successfully injected a message into the network.

Control Registers. NI control registers allows a user process to pass information and commands to the NI device. For example, a processor may want NI interrupts disabled in a critical section. It can do so by writing to a control register in the NI.

Data Registers. NI data registers contain message data sent by a processor or received by the NI from the network.

Notification mechanism. An NI notification mechanism is a mechanism through which the NI informs a process of any change in NI device status. For example, the NI can interrupt the process on a change in device status, such as arrival of a message from the network. Such explicit notification may be unnecessary if a process monitors changes in the NI status registers. Hence, the notification mechanism is optional.

To send a message to the network, a processor first reads the send interface status register to ensure there is enough space in the send interface's data registers. If there is enough space, the processor writes a new message to the data registers. If there is not enough space, the processor can either poll the NI periodically or have the NI notify it when free space becomes available. On receiving the new message in its data register the NI hands the message to the external NI, which injects the message into the network.

When a message arrives at the receiving external NI, the external NI extracts the message from the network and hands the message to the receive interface. The receive interface writes the message to its data registers and sets a status register that indicates to the processor that a message has arrived in the receive interface. Flow control (e.g., return-to-sender) is typically used to ensure messages are never (or rarely) lost if the data registers are full. If the control registers have been appropriately set by the processor, the NI can send a notification to a processor in the receive host node about the arrival of this message through a processor interrupt. Finally, a processor in the receive host node reads the new message from the NI data registers.

Many recent research and commercial computers use a microprocessor instead of a hardware, finite-state machine to run message protocols. The discussion in the rest of this thesis is independent of the presence or absence of any such protocol processor. An internal NI must interact with processors and memory system internal to a node to send and receive messages. Such interaction can use all opportunities for performance improvement that I discuss in this thesis.

1.2 Problems with Conventional Network Interfaces

Conventional internal NI architectures do not efficiently support low-latency communication in a COW. Latency is the user-to-user delay for sending a message. Low latency determines the performance of applications that send small messages frequently. Frequent small messages underlie many traditional and emerging application domains, such as parallel scientific programs [32].

Fortunately, improvements in networks have reduced the impact of the network and software protocols (that generate and consume messages) on the overall latency of communication. In particular, local area network (LAN) bandwidth has improved from 10-100 megabits/second to one gigabit/second or more. Aggressive LANs, such as the Myricom Myrinet [15] or the Tandem Servernet [54], have moved so far that some view them as a new class of networks called a *system area network* or SAN [54, 8] (Appendix A). SANs improve performance in two ways. First, aggressive links and switches provide very high bandwidth and extremely low latency. Second, reliability properties of SANs allow systems to use lean communication layers (e.g., Active Messages [128]) instead of heavyweight and one-size-fits-all protocols (e.g., TCP/IP). Consequently, SANs help improve the performance of both network hardware (links and switches) and network software (communication protocols).

Unfortunately, improvement in network and software protocols have exposed processor-NI interactions as a major bottleneck, particularly for COWs. Conventional internal NI architectures can introduce between ten and hundreds of microseconds latency to the overall communication latency. There are eight factors that can contribute to such latency:

- *Operating system intervention.* Conventional NIs are accessed by the processor via the operating system (e.g., Unix sockets). Such intervention eases protection and address translation for message buffers. Unfortunately, such intervention also introduces long latencies to the critical path of message send and receipt [5, 127]. For example, switching to the operating system from a user process can require execution of between hundreds and thousands of instructions.
- *I/O bus.* Conventional NIs are located on the I/O bus, and not on the higher performance memory bus. This is because I/O buses, unlike proprietary memory buses, offer standard interfaces to which third-party network vendors can manufacture their NI cards. Unfortunately, the latency and bandwidth of I/O buses are two to ten times worse than current memory buses.
- *Small message buffers in NI.* Most conventional NIs offer small amounts of buffering (e.g., few kilobytes) in their internal NIs and rely on processors to buffer messages in main memory. Unfortunately, such processor-controlled buffering can degrade proces-

sor performance by creating resource contention for processors, memory, and system buses.

- *Uncached access to NI registers.* Conventional NIs registers are marked uncachable primarily because such NI designs have side-effects. For example, a load to an NI device register both returns a value and deletes it from the device. Unfortunately, uncached accesses are much slower than memory accesses that hit in processor caches.
- *In-order and non-speculative access to NI registers.* Conventional NIs do not allow processors to access NI registers out of order and speculatively, again, because of the presence of side-effects. For example, if a message send is initiated speculatively, then it cannot be rolled back if the processor later decides that its speculation was incorrect. Therefore, NIs will appear relatively slower, as processors continue to improve via out-of-order and speculative memory accesses.
- *Slow data transfer.* Conventional NIs transfer data from the NI via either uncached loads/stores to memory-mapped NI registers or direct memory access (DMA). Both of these are low-performance solutions. Uncached accesses usually transfer small amounts of data (e.g., between 4 - 16 bytes), thereby offering low bandwidth. DMA transfers large amount of data, but require the operating system to initiate the transfer, which incurs huge latencies.
- *Application Programming Interface (API).* Conventional NIs either directly expose the underlying data transfer primitives (e.g., programmed-controlled I/O via uncached loads/stores to memory-mapped NI registers) or require the operating system to serve as the API (e.g., Unix sockets) for message sends and receives. The first solution offers tight coupling between the processor and NI, which often blocks the processor until the access is complete. The second solution again requires slow operating system intervention. [1]
- *Notification via interrupts.* Conventional NIs usually notify processors of NI events via heavy-weight interrupts. Unfortunately, current microprocessors are not optimized

1. In reality, this interface is somewhere in between an Application Programming Interface (API) and an Application Binary Interface (ABI). I use API due to the lack of a better term.

for interrupts because they treat them as exception conditions. Consequently, interrupt-driven messaging can significantly deteriorate communication performance.

1.3 Future Trends

Technology trends suggest that the latency to access the internal NI will become even more critical in the future. The contribution of software protocols and networks on overall communication latency will continue to reduce. Microprocessors that typically run these protocols are improving at a tremendous pace. The steady drop in feature sizes and introduction of microarchitectural techniques, such as speculative execution [83, 117], are projected to improve microprocessor performance by a factor of 80 in the next ten years [140].

Network performance will keep pace with the improvement in processor performance. Currently, network bandwidth is improving at 100% per year (Figure 2-2), which translates into a factor 1000 improvement in the next ten years. Such massive improvement in bandwidth will make the latency of requests (carrying a few bytes) and responses (carrying multi-megabytes of data) similar. Consequently, overall communication latency will become even more critical. Fortunately, network latencies will also continue to drop in the future because of the advent of low-cost CMOS processes, novel switch architectures [6], and high-speed optical switches [95].

The explosive growth in the performance of microprocessors and networks requires innovative techniques to reduce the latency of processor-NI interactions in a COW. These techniques must also allow a seamless transition towards effectively using future technological and architectural advances. I argue that the solution to this problem is to treat processor-NI interactions like processor-memory interactions (Section 1.4). Since the gap between processor and DRAM memories is increasing [49], microprocessors will continue to invent novel techniques to bridge this gap. In a COW, treating an NI access like a memory access will allow internal NI architectures to take advantage of such future innovations.

1.4 Thesis Contributions for Network Interfaces

This thesis has four contributions. I discuss the first three contributions in this section. Section 1.5 discusses the fourth contribution.

The first contribution of this thesis (Chapter 2) is to show how treating a processor's access to a network interface as a regular memory access can tremendously improve processor-NI interactions in a COW. Memory is virtualized without requiring operating system intervention (in the common case), is on the memory bus, is plentiful in today's computers, does not have side-effects, and hence, can be cached and accessed out of order and speculatively, and is usually transferred in cache block units. I propose to do the same for NIs. Most of the these opportunities have been explored partially and independently by other researchers. My contribution is to organize these opportunities in a single framework that exposes commonality and synergistic interactions.

NIs that use direct memory access (DMA) offer some of these advantages because data DMA-ed into memory can be treated just like regular memory. Unfortunately, the DMA initiation itself often uses high-latency solutions, such as initiation via the operating system (for traditional DMA) or uncached loads/stores (for Princeton's User-Level DMA [11]).

Treating an NI access as a memory access can improve all eight components of latency listed in Section 1.2. Some of these eight opportunities have been partially explored by others. A principal contribution of this work is to organize these opportunities into a common framework. These eight opportunities are:

- using virtual memory hardware, and not operating system intervention, to virtualize the NI (Figure 1-2a),
- placing the NI on the higher performance memory bus, and not on the slower I/O bus,

- using virtual memory as a huge buffer for network messages, instead of small amounts of dedicated memory on the NI,
- caching messages in processor and NI caches, like regular cachable memory,
- allowing out-of-order accesses and speculative loads on a processor’s accesses to an NI, like side-effect-free regular memory accesses,
- transferring messages between processor caches, NI cache, and main memory through cache block transfers, instead of DMA,
- designing the application programming interface (or API) to the NI as memory-based queues, and not directly exposing the underlying data movement primitives as the API, and
- notifying processor of NI events through cache invalidations, instead of heavy-weight interrupts.

The second contribution of this thesis is the design (Chapter 3) and evaluation (Chapter 4) of a new class of network interfaces called *Coherent Network Interfaces (CNIs)*. CNIs sit on a computer’s memory bus and interact with processors via cachable, coherent memory operations. The most aggressive CNI—that is, CNI_iQ_m in my nomenclature—exploits all eight opportunities for performance improvement outlined above.

Chapter 3 develops and optimizes two mechanisms that CNIs use to communicate with processors. A *cachable device register*—derived from cachable control registers [101]—is a coherent, cachable block of memory used to transfer status, control, or data between a CNI and a processor. *Cachable queues* generalize cachable device registers from one cachable, coherent memory block to a contiguous region of cachable, coherent blocks managed as a circular queue. An important advantage of CNIs is that they allow cachable queues to be physically located in processor or CNI caches, but logically allocated in main memory, which allows plentiful buffering. Chapter 3 also explores several critical optimizations—*lazy pointer*, *shadow head*, *sense reverse*, *empty entry removal*, *intra-message prefetch*, *dead message elimination,* and *cache bypass*—that improves the performance of

CNIs. Finally, Chapter 3 examines different alternatives for multiprogramming a CNI that uses cachable queues.

Chapter 4 performs a detailed comparison of four CNIs with a more conventional NI—that is, a Thinking Machines' CM-5 NI [124]—using a 16-node COW, two microbenchmarks, and seven parallel scientific applications. For small message sizes—between 8 and 256 bytes—CNIs improve the round-trip latency by 87-342% compared to a conventional NI on a coherent memory bus. For moderately large messages, between 8 and 4096 bytes, CNIs improved the bandwidth by 109-202%. Results with the seven applications show that CNIs can improve performance by up to 21-190% compared to a conventional NI.

The third contribution of this thesis (Chapter 5) is a systematic classification, examination, and evaluation of two of the eight opportunities—data transfer and buffering—listed above. To the best of my knowledge, this is the first work to systematically identify and explore the data transfer and the buffering parameters that underlie high-performance NIs designed for fine-grain communication. The data transfer parameters capture how messages are transferred between internal memory structures (e.g., processor caches, main memory) of a computer and a memory bus NI. The buffering parameters capture how and where an NI buffers incoming network messages. I evaluate these parameters by comparing seven memory bus NIs using the same system parameters and benchmarks as in Chapter 4. These seven NIs abstract the data transfer and buffering parameters of the NIs in TMC CM-5 [68], Fujitsu AP3000 [109], Princeton User-Level DMA [11], Digital Memory Channel [44], MIT StarT-JR [53], and two Coherent Network Interfaces ($CNI_{512}Q$ and $CNI_{32}Q_m$described in Chapter 4).

My results show that a high-performance NI in a COW should effectively use the block transfer mechanism of the memory bus, minimize processor involvement for data transfer, directly transfer messages between an NI and the processor (at least in the common case), provide plentiful buffering (possibly in main memory), and minimize processor involve-

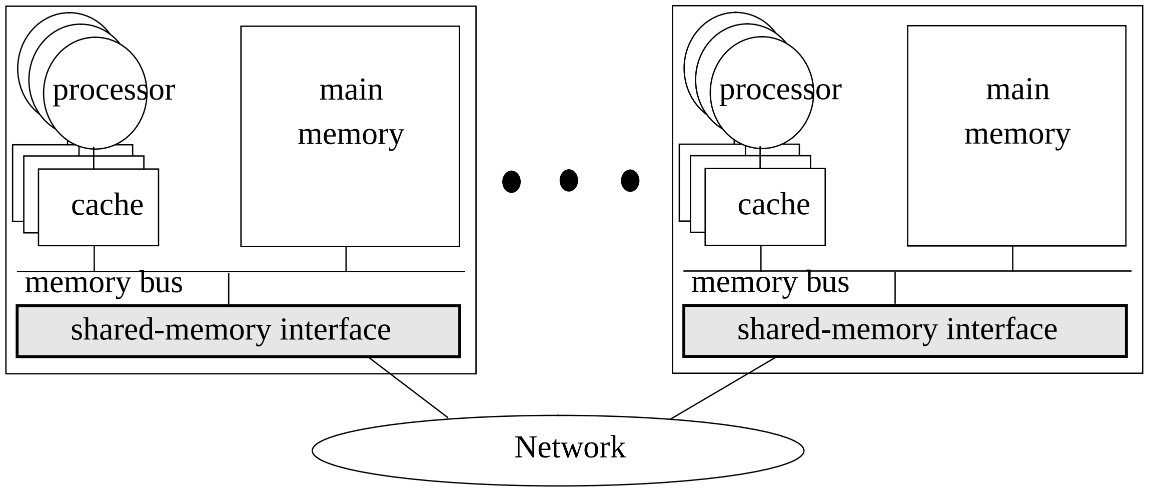


Figure 1-3. A shared-memory multiprocessor.

ment to buffer incoming network messages. $CNI_{32}Q_m$ performs the best among the seven NIs because it optimizes all five data transfer and buffering parameters.

1.5 Using Prediction to Accelerate Coherence Protocols

This part of the thesis differs significantly from the rest of the thesis. Techniques outlined in the rest of the thesis can accelerate user-to-user messaging in a parallel machine programmed with explicit message-passing. In contrast, this section examines techniques to improve the communication performance of parallel machines programmed with a shared-memory programming model.

Shared memory simplifies programming multiprocessors because it provides a single address space to all processors, even when memory is physically distributed among different nodes of the machine (Figure 1-3). To reduce the disparity between latency of local and remote memory accesses, these machines cache both local and remote memory in per-processor caches. Caches are usually made transparent to software with a cache coherence protocol implemented in a shared-memory communication interface. A coherence protocol usually exchanges *coherence messages* between shared-memory interfaces of different nodes to keep per-processor caches coherent. Unlike a message-passing communication interface, a shared-memory interface usually generates these messages directly in hardware or firmware.

Shared-memory interfaces in most large multiprocessors use a form of coherence protocol called a directory protocol [4, 71, 66, 25, 130, 69, 2]. Directory protocols maintain a directory entry per memory block that records which processor(s) currently cache the block. On a processor cache miss for a remotely cached block, the shared-memory interface sends a coherence message over an interconnect to a directory entry, which often forwards message(s) to processor(s) currently caching the block, who then forward data or acknowledgments to the requesting processor and/or directory.

Regrettably, this cache miss and directory activity can disturb a programmer's performance model of shared memory by making some memory accesses tens to hundreds of times slower than others. Ameliorating this problem has led to many proposals, including weaker memory models [1, 43], multithreading [31, 3, 125], non-blocking caches [61, 118], and application-specific coherence protocols [93, 37]. To date, all proposals possess one or more of the following drawbacks: require a more complex programmer interface or model, retard uniprocessor performance, or require sophisticated compilers.

Another class of proposals for ameliorating memory latency is to predict future sharing patterns [9, 46] and take action to overlap coherence message activity with current work. Predictions can be made by programmers [51, 136], compilers [84, 113], or hardware. Specialized predictors in hardware include read-modify-write operation prediction in the SGI Origin protocol [66], pair-wise sharing prediction in SCI [116], dynamic self-invalidation [67], and migratory protocols [28, 120]. Existing predictors, however, are directed at specific sharing patterns known *a priori*. Furthermore, the protocol implementation is often made more complex by intertwining one or more predictors with the standard coherence protocol.

This thesis seeks a more general predictor to accelerate coherence protocols. Predictors would (logically) sit beside each standard directory and cache module to monitor coherence activity and request appropriate actions. If a directory predictor, for example, anticipates that a processor asking for a block B “shared” will next ask for block B “exclusive,” the directory can answer the “shared” request with block B “exclusive.”

The fourth contribution of this thesis (Chapter 6) is the design and evaluation of the *Cosmos* coherence message predictor for accelerating coherence protocols. Cosmos’ design is inspired by Yeh and Patt’s two-level *PAp* branch predictor [139]. Cosmos makes a prediction in two steps. First, it uses a cache block `address` to index into a *Message History Table* to obtain one or more `<processor,message-type>` tuples. These `<processor,message-type>` tuples correspond to sender and message type of the last few coherence messages received for that cache block. Second, it uses these `<processor,message-type>` tuples to index a *Pattern History Table* to obtain a `<processor,message-type>` prediction. Notably, Cosmos faces a greater challenge than branch predictors because the Cosmos’ prediction is a multi-bit `<processor,message-type>` tuple rather than a single bit branch outcome.

My simulation results with five shared-memory applications running on a 16-node parallel machine show that variations of Cosmos predict the source and type of the next coherence message with surprisingly-high accuracies of 62-69% (*barnes*), 84-86% (*moldyn*), 84-85% (*appbt*), 74-92% (*unstructured*), and 84-93% (*dsmc*). Cosmos’ high prediction accuracy results from predictable coherence message patterns or *signatures* associated with specific cache block addresses. Such signatures are generated by sharing patterns [9, 46] that do not change or change very slowly during the execution of these applications.

Chapter 2

Treat Network Interface Access as Memory Access

This thesis argues that NIs should be treated as “standard equipment”, like memory or frame buffer, and not as peripheral add-ons. Today almost every computer is connected to a network. Consequently, every computer needs an NI. Computers connected to low-performance networks may not place a huge performance demand on the NI device. However, NIs for network-centric computers, such a Cluster of Workstations (COW) connected via high-performance networks, such as SANs, must be designed to deliver network performance to host user applications.

I argue that a processor access to an NI in a COW should be treated as a regular memory access, and not as a peripheral I/O operation (e.g., like a disk interface access). Since the gap between processor and DRAM access performance is increasing rapidly, microprocessors will continue to invent novel techniques to bridge this gap. Treating NI accesses as regular memory accesses will allow NI accesses to take advantage of such future innovations. Such treatment opens up at least eight opportunities to improve the performance of processor accesses to the NI [87]. These opportunities are listed in Table 2.1. An NI that supports all the eight opportunities behaves just like another processor cache in an SMP

Problems	**Solutions**		**Discussed**
	Conventional	**Proposed**	
Virtualize via	operating system	virtual memory hardware	Section 2.1
Location	I/O bus	memory bus	Section 2.2
Buffer messages	dedicated memory (in NI or main memory)	virtual memory	Section 2.3
Cache NI registers	not allowed	allowed	Section 2.4
Out-of-order and speculative access	not allowed	allowed	Section 2.5
Message transfer mechanism	DMA or uncached load/ store	cache block transfers	Section 2.6
Application Programming Interface (API)	has side-effects	no side-effects (memory-based queues)	Section 2.7
Notification	Interrupts	cache invalidations	Section 2.8
Summary: NI access similar to	disk interface access	memory access	▨

Table 2.1: Treat NI access as memory access and not as a disk interface access.

node. I discuss these opportunities in greater detail in a survey paper [88].

Many of the eight opportunities listed above have been explored partially by others. A principal contribution of this chapter is to organize these opportunities into a framework that exposes commonality and synergistic interactions. In particular, this thesis examines one opportunity in depth: how caching NI registers can significantly improve the performance of processor-NI interactions.

2.1 Use Virtual Memory Hardware to Virtualize the Network Interface

There is a marked difference in how user processes access a peripheral I/O device (e.g., a disk) and main memory. Both of these are shared physical resources that must be virtualized across multiple user processes. Virtualizing a physical resource to a user process requires two mechanisms: protection and address translation. Protection isolates user processes from one another. Address translation allows a user process to access a physical device through virtual addresses. A peripheral I/O device is virtualized by the OS, which requires all user accesses to I/O devices be initiated through OS traps. Trapping to the OS is usually very expensive because modern microprocessors treat traps as exception conditions, rather than a common occurrence, and hence do not support them very efficiently. Main memory, on the other hand, is virtualized through the virtual memory hardware, which is supported by all high-performance microprocessors today, and does not involve OS intervention in the common case. Main memory is divided into physical pages and mapped to user virtual space on demand. A hardware structure called the Translation Lookaside Buffer rapidly translates user virtual page addresses to physical page addresses in main memory. Consequently, main memory accesses are much faster (less than a microsecond) compared to I/O device accesses (greater than 10 - 100 microseconds).

Accessing NI memory through the virtual memory hardware, and not via the OS, can therefore dramatically improve performance. The OS simply needs to map the NI memory pages directly into user space; the virtual memory hardware that already exists translates these memory-mapped virtual addresses to appropriate physical addresses in the NI memory and ensures protected access to it.

The TMC CM-5 NI and, more recently, the Myricom Myrinet host interface allow users to directly access the NI memory using this technique. I call such NIs *User-Level Network Interfaces (ULNIs*) since the NI memory can be directly accessed from user space. Compaq Corporation, Intel Corporation, and Microsoft Corporation are jointly developing such a ULNI specification called the Virtual Interface Architecture [36]. The VI architecture is a logical specification that will allow a user process to directly access the internal NI memory and thereby bypass the operating system to send and receive messages from the network.

2.2 Place the Network Interface on the Memory Bus

In a standard workstation node (Figure 2-1a), I/O devices are typically located on the peripheral I/O bus. The choice of this location is dictated primarily by the availability of a standard I/O bus interface (e.g., SBus, PCI), which enables independent vendors to manufacture NI cards to these standard specifications. Unlike I/O buses, current memory buses

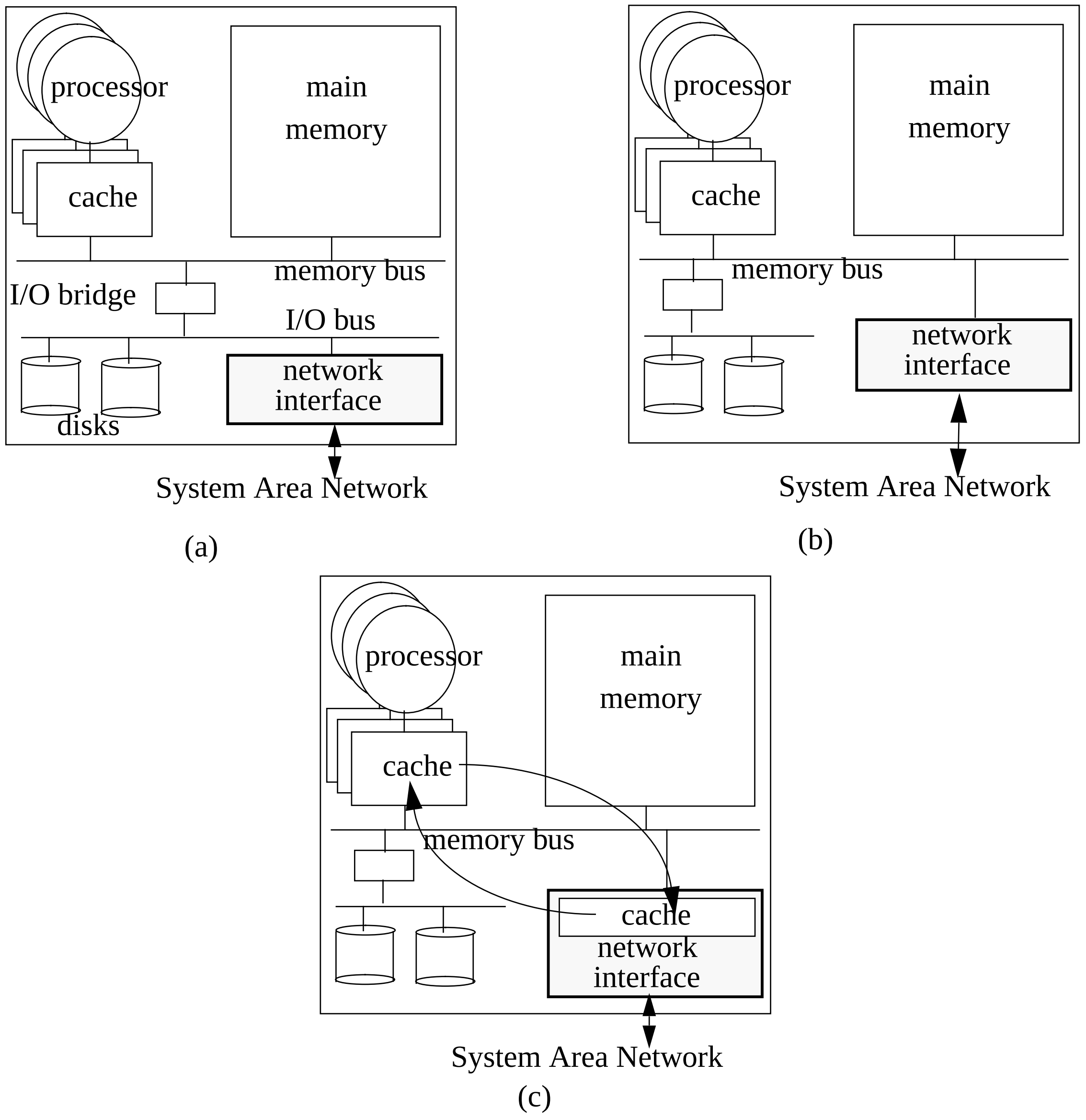


Figure 2-1. Workstation Nodes with Network Interfaces. (a) shows the architecture of standard workstation node with the network interface on the I/O bus. (b) shows the same workstation node with the network interface on the memory bus. (c) augments the network interface in (b) with a cache.

are usually proprietary, have non-standard interfaces because they often change across processor generations, and hence, manufacturers of I/O devices do not usually design I/O devices to memory bus specifications.

Current memory buses, however, offer three significant performance advantages over I/O buses. First, memory buses are much faster because they are typically clocked at a higher frequency compared to I/O buses. For example, current PC memory buses are clocked between 66-75 MHz, which is more than two times faster than the current generation of 33 MHz PCI buses. Typically, all I/O bus accesses also additionally traverse the memory bus and an I/O bridge that connects proprietary memory buses to standard I/O buses.

Second, memory buses offer significantly higher bandwidth than I/O buses. Current PC memory buses offer peak bandwidths greater than 400 megabytes/second. This is more than four times greater than the peak bandwidth offered by the current generation of PCI buses. Some of the Sun Enterprise servers support an even more aggressive memory bus called the UltraGigaplane, which offers a sustained bandwidth of 2.6 gigabytes/second. Memory buses can offer such high bandwidth because these buses are 64- to 256-bits wide, which is a factor of two to eight greater than current 32-bit wide PCI bus. Additionally, today’s memory buses support split transactions, which improves bandwidth because a device or a cache no longer has to lock down the memory bus during the entire duration of a transaction (e.g., cache block read from main memory).

Figure 2-2 suggests that the gap between of bandwidths of memory and I/O buses will continue to exist in future. In fact, I/O bus bandwidth lags behind memory bus bandwidth by at least five years. In other words, I/O buses will take another five years to achieve the peak bandwidth offered by today’s PC memory buses. Consequently, NI cards designed to I/O buses will not be able to harness the full memory bus bandwidth. Figure 2-2 also shows that SAN link bandwidth is growing at a much faster rate than the bandwidth of PC memory buses. For such SANs we will need more aggressive memory buses, such as the SUN Ultragigaplane.

Third, memory buses support optimized single-writer coherence protocols, which allows processor caches to easily cache and share memory. This is because these single-writer

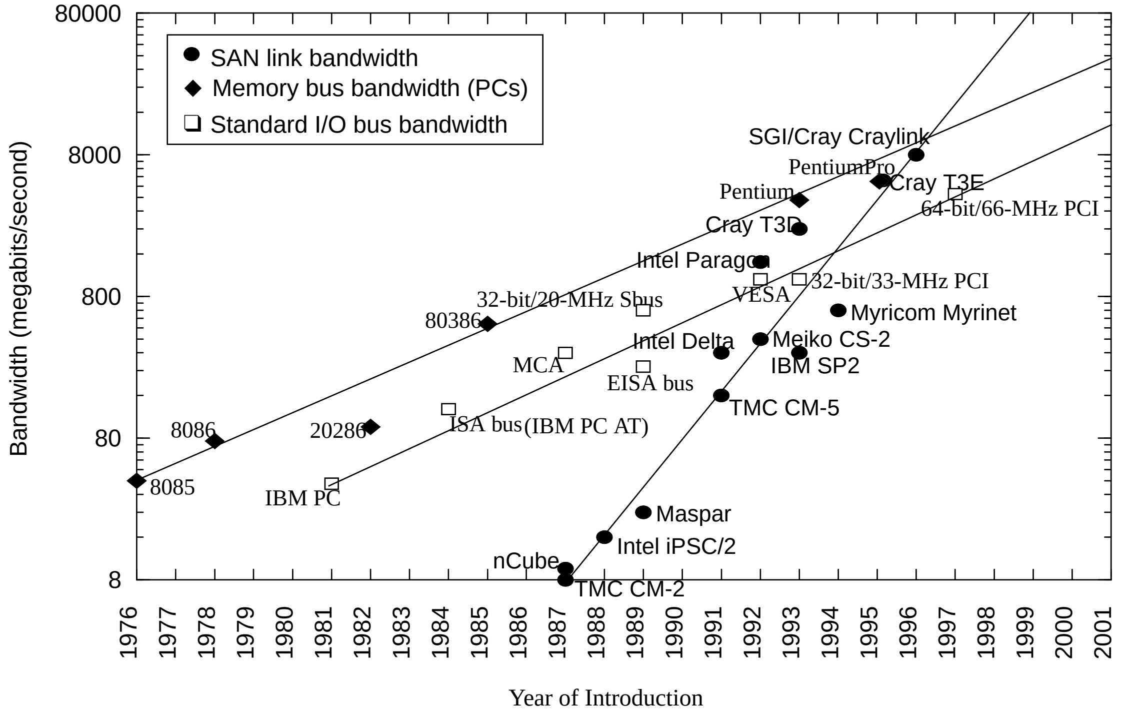


Figure 2-2. Trends in peak SAN link bandwidth and I/O bus bandwidth. This figure shows the crossover point between peak System Area Network (SAN) link bandwidth and “standard” I/O bus bandwidth. SAN link bandwidth has been increasing by roughly 100% per year, while the I/O bus bandwidth has been increasing by roughly 32% per year. SAN references: Cray T3E [106, 102], SGI/Cray Craylink [41, 45], Myricom Myrinet [15, 56], and the rest from Figure 7.19 (Page 591) of Hennessy and Patterson’s book [49]. Memory bus references: Polsson’s article on History of Microcomputers [98]. I/O bus references: 32-bit/20-MHz SBus [55], 64-bit/66-MHz PCI [77], and rest from Needham’s article on PCI [94].

coherence protocols provide a single and consistent image of physical memory across all processor caches. Section 2.4 shows how and why caching message data in processor and ULNI caches can help improve performance.

The performance advantages of memory buses strongly suggest that ULNIs should be placed on memory buses, just like main memory (Figure 2-1b). My simulation results in Chapter 4 with several parallel scientific applications confirms the performance advantages of memory bus NIs over I/O bus NIs.

The only disadvantage of current memory buses is that they do not usually export a standard interface to which independent vendors can design ULNIs to. However, the advent of

ULNIs as “standard equipment”, like memory or frame buffers, emphasizes the need for memory bus designers to export a standard interface to either systems designers internal to a company or third-party vendors manufacturing independent ULNI devices. Companies, such as Intel, IBM, and Sun Microsystems, that manufacture both microprocessors and network-centric computers can allow system designers to design ULNIs to their internal memory bus. Intel’s MPP supercomputer called Teraflop [19], for example, attaches the ULNI device directly on the PentiumPro memory bus. For independent vendors finding a standard interface on the memory bus may imply coordinating with microprocessor companies to get access to their memory bus specification.[1] Alternatively, manufacturers of proprietary memory buses could provide special *bridges* to other open standard interfaces, such as the PCI interface.

The bridge we need converts proprietary memory bus signals to and from another specification. A standard bridge might connect to a standard I/O bus, such as PCI. A standard bridge supports many standard devices. However, it may not provide the performance or coherence access needed by ULNIs. A more aggressive bridge could convert directly to a standard I/O bus connector that supports one demanding I/O device without a physical I/O bus. This bridge can fake the I/O bus signals to offer higher performance (e.g., no arbitration time) to standard devices. The SGI Power Challenge, for example, uses this type of bridge (which they call a “personality interface”) to convert between their proprietary I/O bus and a standard SCSI device. Similarly, Intel’s Accelerated Graphics Port [138] is a standard bridge that offers graphics accelerators a dedicated high-bandwidth path to main memory. An even more aggressive bridge can convert to a device-specific interface that is proprietary, but less demanding and more stable between product generations than a memory bus. If network connections become “standard equipment” like frame buffers, this option provides an attractive way to obtain nearly the network performance of a memory-bus ULNI without some of the cost.

1. Corollary, Inc. obtained access to the PentiumPro memory bus to build a shared-memory system called Profusion [126].

Yet another possibility would be to standardize the interface between the internal and external NIs (Section 1.1). Microprocessor vendors can provide the internal interface that communicates with the processor and third-party vendors can provide the external interface that talks to the network. This would relieve third-party vendors from having to worry about the details of a particular memory bus' coherence protocol and allow microprocessor vendors to deliver the network's performance to a user process via its own optimized internal interface.

2.3 Use Virtual Memory to Buffer Network Messages

Peripheral I/O devices may require large amounts of memory. For I/O devices such as 3D Graphics Accelerators, in particular, the demand for memory is increasing steadily because high quality images and image transformations require large amounts (e.g., tens of megabytes) of primary storage. The Accelerated Graphics Port (AGP) [138] was designed to counter this demand. AGP provides graphics devices with a standardized high bandwidth path to main memory, which allows graphics devices to use main memory as a large graphics buffer. This enables low-cost graphics devices because dedicated memory can incur a prohibitive cost for graphics accelerators. Because 3D Graphics accelerators can now use portions of system memory through a graphics-specific dedicated path, these accelerators have become less peripheral in nature.

Like 3D Graphics Accelerators, high-performance ULNI devices can require large amounts, that is, tens of megabytes, of memory to buffer outgoing and incoming network messages. This is because of four reasons. First, variation in performance of loosely-coupled microprocessors and SAN switches and advent of a variety user-level communication protocols often create a temporary mismatch between the rates at which network messages are generated, transferred, and consumed. Buffering smooths out these rates and helps create a balanced system.

Second, with limited buffering and bursts of messages—a common occurrence in loosely synchronized parallel applications—a processor must constantly monitor ULNI status changes and remove messages from the limited ULNI buffers to avoid clogging up the network. This can significantly degrade performance if the processor must continuously read an uncached status register (Section 2.8).

Third, a limited amount of ULNI buffering severely restricts the degree of multiprogramming because these ULNI buffers must be divided among different processes. Alternatively, the operating system can switch the buffers among processes; but, this can be a very expensive operation.

Fourth, SANs, such as the Myricom Myrinet, requires ULNIs to perform some form of flow-control, such as all-to-all buffer reservation or return-to-sender, to guarantee end-to-end reliable message delivery. To avoid clogging the network, such flow control schemes may require large amounts of ULNI buffering.

Current commercial ULNIs, such as the Myricom Myrinet's host interface, provide only around hundreds of kilobytes of message buffers in the ULNI. This amount is not enough to support large systems with a large degree of multiprogramming. Fortunately, the Myricom Myrinet host interface provides a microprocessor that can be programmed to overcome this problem, as outlined below.

The problem of limited buffering in the ULNIs can be solved by buffering network messages in the user's virtual space [72]. This provides large amounts of buffering limited only by the size of main memory (and swap space, which backs up the user virtual space). A ULNI with this capability, however, requires additional support for protection and address translation, which may require moderate to substantial changes to commodity operating systems. The problems are similar to those faced by SMP nodes today. Just like a processor, the ULNI must have access to virtual-to-physical address translations. To reduce the cost of accessing these translations every time a message is retrieved from or deposited into main memory, a ULNI can cache these translations in a structure similar to a processor TLB. ULNIs must, therefore, be prepared to service ULNI TLB misses as well

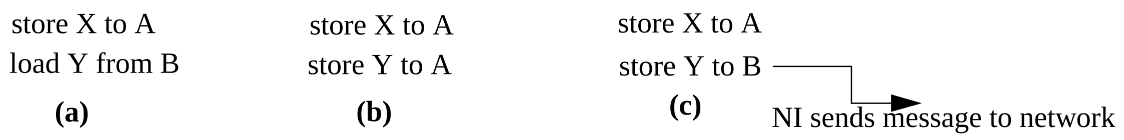


Figure 2-3. Three examples of side-effects in existing NI designs. The instructions shown in this figure are uncached loads and stores to ULNI registers memory-mapped to virtual addresses A and B. (a) shows that the store-load pair must be strictly in order for some NIs to work correctly (e.g., Princeton UDMA initiation), even though the instructions appear unrelated to the processor. (b) shows two consecutive stores to the same address must occur in order (e.g., TMC CM-5 NI). (c) extends (b) to show that the second store (in the general case, the "n"th store) can trigger an action in the NI, such as sending a message into the network. (a) and (b) have the side-effects that a previous store determines implicitly the next uncached load or store a NI expects. (c) has the side-effect of sending a message on a store.

as invalidate or update the TLB entries when the OS remaps a page or swaps it out to disk. However, servicing TLB misses is a complex operation because it can result in page faults, is often processor-specific, and may require OS intervention. A ULNI can reduce this complexity by interrupting and requesting the OS running on the host processor to insert the appropriate translation in the ULNI TLB. The ULNI TLB can be augmented with protection bits and process identifiers, similar to those in a modern processor TLB, and updated along with the translations to ensure protected ULNI access to main memory. Other researchers have explored these issues in detail [133, 47, 105].

Chapter 5 evaluates the impact of alternate buffering strategies on seven parallel scientific applications.

2.4 Cache NI Registers in Processor and NI Caches

Unlike main memory, peripheral I/O device memory, such as ULNI memory, is typically not cached in processor caches. Instead, ULNI memory is marked uncachable. This is because of three reasons. First, processor accesses to ULNI device memory often have *side-effects* (Figure 2-3) that force all such accesses to be visible to the ULNI. Hence, all processor accesses must be uncached because cached loads or stores may not be visible outside the processor, unless there is a cache miss.

Second, a ULNI device must generate coherence signals to invalidate messages in processor caches. In this respect, it behaves more like a processor cache, rather than main memory. When a message arrives and the ULNI writes new data to the message buffers, the corresponding memory locations for the message buffers in processor caches must be invalidated. Otherwise, processors can read stale data from these buffers. Unfortunately, many ULNIs reside on standard I/O buses, which usually do not support such coherence signals. Hence, ULNIs usually do not allow processors to cache ULNI memory.[1]

Third, caching ULNI registers in processor caches require extra support for ULNI register reuse. Conventional ULNI device registers solve the problem of register reuse using implicit *clear-on-read* semantics, where the register is cleared after an uncached load. For example, the CM-5 NI treats the read of the hardware receive fifo as an implicit "pop" operation. Clear-on-read works because processors guarantee the atomicity of individual load instructions; that is, the value returned by the device is guaranteed to be written to a register. Clear-on-read does not work well for cached ULNI registers, since most processors do not provide the same atomicity guarantees for cache blocks. Processors guarantee the load that causes the cache miss to be atomic to ensure forward progress; however, there are no guarantees for the remaining words in the block. Before subsequent loads complete, a cache conflict (e.g., resulting from an interrupt) could replace the block. With clear-on-read semantics, the remainder of the data in the cache block would be lost forever. [2]

The first and third problems—the presence of side-effects in ULNI memory accesses and the absence of clear-on-read semantics on cache blocks—can be eliminated by designing the application programming interface carefully (Section 2.7). The second problem—keeping ULNI device memory and processor caches coherent—can be solved by

1. This is different from the standard problem of "coherent I/O." A computer with coherent I/O allows processors to cache I/O space. Here, I am interested in the inverse problem, that is, caching memory space in an I/O device.
2. This problem may not arise if only one NI register (that can be accessed via a single load or store) is allocated per cache block because individual cached loads are usually guaranteed to be atomic.

placing the ULNI device on the memory bus, so that a ULNI can directly observe and participate in the system's coherence protocol, just like a processor cache in an SMP.

Caching ULNI registers in processor caches offers two advantages. First, caching status or control registers in processor caches helps remove unnecessary memory bus traffic. For example, if a processor were polling on an uncached status register, every processor poll would go across the memory bus to the ULNI device. In the absence of any message in the ULNI, unsuccessful polls that do not find a message in the ULNI device waste precious memory bus bandwidth, which could be used by other processors in an SMP node. Instead, if the processor polls on a cached memory location, which contains the ULNI status information, all unsuccessful polls will hit in the processor's cache. When a message arrives finally and the ULNI status changes, the ULNI device invalidates the cached status register in the processor's cache. On its next poll attempt, the processor will incur a cache miss, which can be satisfied directly by the ULNI.

Second, uncached accesses provide very low bandwidth compared to cache block accesses because they transfer only a few bytes of data (e.g., 1-16 bytes). In contrast, cache blocks are typically much larger (e.g., 32-128 bytes). Hence, they can exploit the full transfer bandwidth of today's memory buses. Section 2.6 discusses these issues in detail. Table 4.5 shows that a large fraction of the memory bus bandwidth can be used effectively for processor accesses to NI registers.

Like processor caches, ULNI caches can cache ULNI registers as well. Instead of allocating ULNI registers in ULNI memory, the registers can be allocated in the user's virtual space and backed up by main memory. Section 3.6 discusses details of how such registers can be mapped, accessed, and synchronized among different application processes. Like processor caches, ULNI caches can simply cache the portion of main memory that contains the ULNI registers. Such ULNI caches help improve performance in three ways. First, processor cache misses for ULNI registers can be intercepted and satisfied directly by the ULNI cache through a direct cache-to-cache transfer. Contrast this with data transfer via DMA in which messages reach the processor cache in two steps (and, consequently two memory bus crossings): from ULNI device to main memory and from main memory to the processor cache. This increase in latency may become critical for latency-bound, request-response protocols.

Second, when bursts of messages arrive at an ULNI, the ULNI cache may overflow; but, ULNI cache replacements to main memory will automatically buffer these messages without any processor intervention. Contrast this with register-mapped ULNIs in which processors must explicitly copy the data from the ULNI registers to the user's virtual space, which can severely degrade performance.

Third, communication protocols, such as an update protocol in a software distributed shared-memory architecture, often package the same data block in different messages and send them to different host nodes over the network. In the absence of a cache, for each message sent the processor must explicitly write the data block into the ULNI or the ULNI must fetch the data block from the processor's cache or main memory. With a cache, the data block needs to be transferred from the processor's cache or main memory only the first time. Subsequent ULNI accesses to the data block will hit in the ULNI cache. This advantage may not be reaped, however, if users explicitly copy the data block from user space to ULNI data structures (e.g., ULNI queues) for each message. To make this scheme effective, the user application programming interface must allow users to specify the user virtual address to data blocks residing in user virtual space (see Section 2.7). The virtual address allows the ULNI to determine that the data block is cached in the ULNI, and therefore, need not be fetched again.

Chapter 3 explores the above issues in detail and proposes mechanisms that allow a processor and an NI to cache NI registers.

2.5 Allow Out-of-Order and Speculative Accesses to NI Memory

To tolerate the latency of main memory access, processors allow loads and stores to bypass earlier loads or stores. This is called out-of-order (OOO) memory access. Four key system changes have taken place to realize out-of-order accesses to main memory: processors can issue out-of-order loads and stores, caches are non-blocking so that they do not stall on consecutive cache misses, memory buses can support several outstanding requests to main memory, and finally, the memory controller can handle multiple requests to the memory system.

Speculative execution is more aggressive than OOO accesses in tolerating memory access latency. Processors speculate on control dependence (e.g., branch prediction), data dependence, data addresses, and data values, and perform computations based on these speculated values. If the speculation is successful, idle processor resources can be used effectively and memory access latencies can be tolerated. However, if the speculation is incorrect, then all previous computation based on speculatively loaded values must be squashed and any process-specific state must be rolled back to the point from where the speculation failed. In the context of messaging, I want processors to speculatively read from and write messages to ULNI memory, just like regular memory.

Processors do not usually perform OOO and speculative accesses to I/O device memory because of three reasons. First, many I/O buses do not adequately support multiple outstanding transactions, which forces processor accesses to I/O device memory to be serialized on the I/O bus. Second, the presence of side-effects (Section 2.4) in I/O devices often force I/O device memory accesses to be performed in order, which prevents OOO accesses to I/O device memory. Further, current I/O devices do not provide any mechanism to rollback any side-effects if the processor's speculation is incorrect, which prevents speculative loads to I/O device memory. Third, the most microprocessors today disallow OOO and speculative accesses on uncached loads or stores, which is the predominant way in which I/O devices are accessed today.

The first problem—the absence of support for multiple outstanding transactions on common I/O buses—can be solved by interfacing the ULNI device to the memory bus, which usually supports multiple outstanding transactions. The second problem—presence of side-effects in ULNI memory accesses—can be eliminated by designing the application programming interface to the ULNI carefully (Section 2.7). Finally, the third problem—absence of OOO and speculative access to uncached I/O space—can be solved be caching ULNI registers in processor caches, because modern microprocessors can perform OOO and speculative accesses from regular cachable memory (Section 2.4), and not allowing speculatively stored state and memory to be reflected outside the processor, which most speculative processors already support.

Chapter 6 looks even further into the future and proposes a coherence message predictor called *Cosmos*. Cosmos allows processors to speculatively send, receive, and process coherence messages in a cache-coherent, shared-memory machine.

2.6 Move Data Between a Processor and an NI in Cache Block Units

Processors typically exchange data with peripheral I/O devices via uncached loads or stores or DMA. In contrast, processor accesses to main memory is typically satisfied through cache misses; data is transferred in cache block units from main memory to processor caches.

Both uncached accesses and DMA are potentially low performance solutions for data movement between processors and peripheral I/O devices. Uncached loads or stores transfer only a few bytes of data (e.g., 1-16 bytes). This wastes bus bandwidth because most modern memory buses are wider than 16 bytes (e.g., SUN UltraGigaplane is 32-byte wide). Uncached accesses can, however, become viable for data movement if processors provide special support, such as coalescing store buffers to merge multiple uncached stores to device registers (e.g., MIPS R10000) or special instructions to move chunks of device data to FP registers (e.g., Sun UltraSPARC). Unfortunately, these mechanisms are processor-dependent and limited in scope, so third party vendors cannot always rely on such support to design their ULNIs.

Unlike uncached accesses, DMA engines can use the full transfer bandwidth of memory buses. However, transferring data between processors and I/O devices via DMA can still be expensive because of two reasons. First, traditional DMA is initiated through the OS, which incurs very high initiation overhead. Second, at the receive end, DMA-ed data reaches a processor cache through two hops—one hop from the ULNI device to main memory and the second hop from main memory to the processor cache. Princeton's UDMA dramatically reduces the DMA initiation overhead on the send side by allowing processors to initiate DMA directly through a two-instruction sequence from user space without OS intervention. Reinhardt, et al. [101] demonstrated that UDMA can be used as cheaply on the receive side as well. Unfortunately, the UDMA initiation scheme suffers from side-effects (Section 2.4) and, like traditional DMA, transfers data in two hops.

Transferring data between processor caches and ULNIs through cache block transfers, just like regular memory, combines the benefits of uncached accesses and DMA. Like uncached accesses, data can be transferred directly from the ULNI memory to the processor without an extra hop through main memory. Like DMA, cache block transfers can fully use the memory bus transfer bandwidth, particularly because today's wide memory buses are optimized for cache block transfers. In fact, current DMA engines transfer data over the memory bus using coherent, cache block transfers to avoid having stale data in processor caches when new data is DMA-ed into main memory.[1] Additionally, transferring data in cache block units does not preclude overlap of computation and communication (see Section 2.7).

Chapter 5 explores the impact of alternate data transfer strategies on the performance of seven parallel scientific applications.

1. Transferring messages from the ULNI to the processor cache may, however, be wasteful if the application simply intends to transfer data from the network interface to another I/O device (e.g., disk or graphics buffer).

2.7 Use Memory-Based Queues as Application Programming Interface

Typically, a user process accesses a peripheral I/O device via the OS or uses the underlying data movement primitive as its Application Programming Interface (API) to the I/O device. For example, user APIs based on program-controlled I/O expose uncached loads and stores—the data movement primitives—to memory-mapped device registers as the user API to the I/O device. Similarly, Princeton's UDMA mechanism exposes DMA transfers as the user API to the ULNI device. In this section I argue that instead of exposing the underlying data movement primitive as the user API, ULNIs should structure the ULNI date registers as *memory-based* queues [35, 127].[1] Such memory-based queues can be classified neither as program-controlled I/O nor as DMA. I believe memory-based queues are a natural and simple extension to the hardware FIFOs used in many NIs. More complicated memory-based structures can also work as long as they can be structured to avoid side-effects.

Memory-based queues consist of two parts: a send queue and a receive queue. Each queue is allocated in virtual memory and managed as a circular buffer with head and tail pointers. To send a message, the processor enqueues the message at the tail of the send queue either by explicitly writing the message into the send queue memory or by inserting a virtual pointer to the message into the send queue. The ULNI dequeues messages from the head by reading the send queue memory and, if necessary, translating the virtual pointer to the message to its physical memory address (Section 2.3) and subsequently reading the message from the user virtual space. For the receive queue, the ULNI similarly enqueues messages at the tail of the receive queue and the processor dequeues messages from the head. Device commands for such APIs are no longer explicit DMA-initiation requests; instead, ULNI device commands are simple memory operations, such as incre-

1. Brewer, et al. [16] and Scott [107] have proposed and implemented a different type of memory-based queue, which can be allocated, controlled, and programmed directly in user space. An NI may or may not be aware of the presence of such a queueing structure. In contrast, the memory-based queues I describe here serve as communication channels between a user process and an NI.

menting or decrementing queue head or tail pointers. For example, when a processor enqueues a message to the send queue and increments the tail pointer, the ULNI interprets this as a device command to send a message out to the network. If the tail pointer is uncached, then the ULNI treats the increment as a signalling store; if the tail pointer is cached, the ULNI must poll on the tail pointer for new messages. Section 3.6 discusses how such queues can be accessed by and synchronized among different processes.

Memory-based queues can be extended to support zero-copy protocols that place data directly into a user’s data structures. Instead of writing the message to the memory-based queue, the processor or ULNI could write the virtual address of the message data to the queue location. This creates additional complexity in the ULNI because it must now translate the virtual address to its corresponding physical address to obtain the message data. Section 3.6.2 discusses how a ULNI can obtain such physical addresses.

There are four advantages to treating ULNI API as memory-based queues. First, unlike uncached accesses or UDMA, memory-based queues decouple a processor and a ULNI, which enables both the processor and ULNI to send and receive multiple messages to and from the queues without blocking. Additionally, unlike uncached accesses, but like DMA or UDMA, memory-based queues allow overlap of a processor’s computation with data transfer to and from the NI.

Second, memory-based queues avoid side-effects by treating ULNI queue accesses simply as side-effect-free regular memory accesses. Thus, ULNI queues do not require in-order accesses, do not provide a single fixed address for the entire queue, and separate ULNI queue memory access from ULNI commands, because ULNI commands now are mostly incrementing or decrementing queue pointers. This allows processors to cache ULNI queues, perform OOO accesses on queue memory, and speculatively send and receive messages to and from these queues. Speculative sends work because modern microprocessors buffer all speculatively-stored memory (in our case, messages) internally within the processor. If and when the speculation succeeds, the processor flushes these buffers into regular cachable memory locations (in our case, memory-based queues).

Third, since memory-based queues are allocated like regular memory and managed as circular buffers, the reuse handshake is simple: a comparison of the head and tail pointers reveals whether a queue location can be reused or not. Even the reuse mechanism can be optimized through techniques, such as lazy pointers, message valid bits, sense reverse, and empty entry removal (see Chapter 3). This simple reuse handshake makes caching ULNI data registers much easier.

Fourth, memory-based queues simplify the problem of multiprogramming a ULNI for SMPs. In an SMP multiple processes running on different processors can simultaneously access the ULNI device. This simultaneous access makes the multiprogramming problem much harder. This is because simple solutions, such as the one adopted by the CM-5 in which the ULNI and user process are context switched together, are no longer feasible. Memory-based queues offer a more elegant solution: each user process negotiates its own private communication channel with the ULNI device through memory-based ULNI queues mapped to user virtual address space (see Section 3.6 for details). This allows each user process simultaneous, but protected, access to the ULNI. This method does, however, involve two complexities. First, the ULNI must now multiplex the internal and external NI ports among these queues. Second, since ULNIs can only support limited amounts of memory on the device, ULNIs must be prepared to context-switch the ULNI queue state (e.g., head and tail pointers) if they reside in dedicated ULNI memory. However, if the ULNI memory is treated as a cache (Section 2.4), then ULNIs do not have to explicitly manage these queues and context-switch them. This is because the queues are automatically displaced to main memory when the ULNI cache overflows and reinserted into the ULNI cache through ULNI cache misses.Chapter 3.6 explores these issues in more detail.

Using memory-based queues (or, more generally, treating an NI access as regular memory access) raises an important concern about the error model seen by the user. This is

Error		Network	Memory
Corruption	Detection	Parity, Checksum, CRC	Parity, ECC
	Recovery	Link-level error correction	Error Correction
		Retransmit	User memory: kill process Kernel memory: reboot
Loss	Detection	Timeout	Timeout
	Recovery	Retransmit	Reboot

Table 2.2: Error model for networks and memory.

because the error model for networks and memory have been different traditionally (Table 2.2).

Networks can incur two important types of errors: link error and end-to-end error. Link errors can be as high as one corrupted bit in 10^{12} bits, which translates into one bit error per 16.67 minutes for a gigabit-per-second network link. System Area Networks (Appendix A) typically cope with this problem using CRC (cyclic redundancy check) and flow control (to prevent buffer overflows in switches).

Prevention of link-level error also drastically reduces the possibility of end-to-end errors. Consequently, traditional solutions, such as those used in software protocol stacks for local area networks, may be overkill. Instead, processors can partially or fully adopt the error model for memory in which memory corruption in certain situations can force a reboot of the system. Thus, on detecting an error, an NI can simply signal a error to the host's operating system, which can either decide to reboot itself, crash the user process, or simply flag an error to the user process.

Alternatively, on detecting a network message corruption or loss, a sender could still retransmit a message. Memory-based queues offer an easy solution for this. The sender does not have to free up the queue position corresponding to a message until it knows for sure that the receiver has successfully accepted that message. If the sender detects message corruption or loss, it can simply retransmit the message from the queue position allocated for the message.

For my simulations in this I make the following assumptions:

- Transient failures in the network links are rare because of link-level CRC checks and flow control. If an error is detected by the CRC in the NI, then the NI flags an error to the operating system signalling a fatal crash and perhaps a system reboot.
- The network switches do not drop packets on congestion. Instead, they back up the network.
- The NI does not drop messages if its buffers are full. I use a flow control scheme called return-to-sender in which the receiving NI returns an incoming message to the sender if it does not enough buffer space. The sender must be able to sink the message and retransmit it later. The sender sinks the message by preallocating buffer space before it sent the message. Consequently, when the receiver accepts a message, it must send an acknowledgment to the sender to free up the buffer space.

2.8 Use Cache Invalidations as Notification Signals

Peripheral I/O devices, such as disks, have typically notified user processes of changes in I/O device status through two mechanisms. They interrupt the user process when the I/O device status changes (e.g., in Unix this is done through the signal interface). Alternatively, some I/O devices allow a user process to monitor changes in I/O device status by polling on an uncached memory-mapped status register. Unfortunately, notification through either interrupts and or polling on uncached device registers is expensive.

Notification through interrupts is very slow because of three reasons. First, these notifications must be vectored to the user process through the operating system, which executes hundreds of instructions before the interrupt is delivered to the user. Second, switching back and forth between a user process and the operating system pollutes the processor’s hardware structures, such as the instruction cache, data cache, TLB, and branch prediction table. Third, these interrupts force today's OOO and speculative microprocessors (Section 2.5) to stop their OOO and speculation engines. However, a few modern microprocessor architectures do provide a few hooks to improve the performance of interrupts. For example, SPARC Version 9 has added eight scratch registers for interrupt handlers. Of course, microprocessors can be designed to more efficiently support interrupt handlers. Nevertheless, vectoring an interrupt through the operating system is and will continue to be expensive because interrupts are treated both by the processor and operating system as an exception condition, and not a common occurrence.

Alternatively, processors can poll on uncached memory-mapped device registers to monitor changes in I/O device status. Polling is cheaper than interrupts because poll instructions—uncached loads—can be issued directly from user space. However, polling can be harmful if the frequency of polling is significantly higher than the rate at which messages arrive. Additionally, polling on uncached status registers can waste precious memory bus bandwidth (Section 2.4).

As described in Section 2.4, the cost of polling can be reduced significantly if a processor treats the ULNI status register as part of regular cachable memory. In the absence of a message, a processor's accesses to the status register will repeatedly hit in its cache. A processor can poll on a status register using two methods. The processor can poll on a regular cachable memory location, which is updated by an operating system interrupt handler. The interrupt handler is triggered via a ULNI interrupt. Alternatively, the processor can poll directly on a ULNI status register.When the ULNI status changes, the ULNI simply invalidates the status register in the processor's cache, which serves as a ULNI notification signal to the processor. The processor simply reads the new status via a cache miss satisfied directly by the ULNI. The cost of this cache miss can be further amortized by pulling in part of the message in the cache block along with the status register itself.

2.9 Conclusion

A new generation of networks called System Area Networks (SANs) has evolved to satisfy the increasing demand for high-bandwidth, low-latency networks. The benefits of SANs are realized in applications only if light-weight protocols (not TCP/IP) and efficient network interfaces are used. The benefits of SANs are squandered, for example, if applications must invoke the operating system to send and receive messages. In contrast, User-level Network Interfaces (ULNIs) allow host applications to access the network interface directly without compromising protection by memory mapping internal interface registers into user space.

Future trends such as the exponential improvement in microprocessors' and SANs' performance and the advent of SMPs indicate that processor accesses to ULNIs will become a critical bottleneck for computer systems built with SANs. Processor accesses to ULNI registers is simply reading and writing ULNI memory. Nevertheless, most ULNIs treat such accesses as peripheral I/O operations that can have side-effects (e.g., a message send). Such treatment disallows current ULNIs to take advantage of memory access optimization techniques such as traditional caches, out-of-order accesses, and speculation.

I have argued that ULNI memory accesses should be treated as regular side-effect-free memory accesses and not as peripheral disk I/O operations to improve processor accesses to ULNI registers. Such treatment allows eight opportunities to improve performance of ULNI accesses. First, virtual memory hardware should be used to virtualize the NI. Second, the ULNI should be placed on the high-performance memory bus. Memory bus interfaces often change over processor generations. Consequently, microprocessor vendors are often reluctant to export their memory bus interfaces to independent vendors. Nevertheless there are several alternative ways in which an independent vendor can place their ULNI cards on the memory bus. Third, messages should be cached in processor and ULNI caches. Fourth, out-of-order accesses and speculation on processor accesses to an ULNI should be allowed. Fifth, the application programming interface (API) to the ULNI should be designed as memory-based queues. Sixth, virtual memory should be used to buffer network messages. Seventh, messages should be transferred between processor caches, ULNI caches, and main memory through cache block transfers. Finally, the processor should be notified of ULNI events through cache invalidations.

Many of the eight opportunities discussed in this chapter have been explored partially and independently by other researchers. A principal contribution of this chapter is to organize these opportunities in one common framework. Chapter 3 and Chapter 4 examine one particular opportunity in detail. That is, how processor accesses to an NI can be improved significantly by allowing an NI to interact with the rest of the system via coherent, cachable memory operations.

Chapter 3

Coherent Network Interfaces Techniques

This thesis argues that a processor's accesses to ULNI (User-Level Network Interface) registers should be treated as regular, side-effect-free, memory accesses. The previous chapter examined eight opportunities for optimization exposed by such treatment. This chapter develops and examines specific mechanisms for a class of ULNIs called Coherent Network Interfaces (CNIs), which interact with the processor via the node's coherence protocol. The optimal CNI design effectively uses all eight opportunities for optimization. Additionally, all CNI mechanisms described in this chapter work with, and require no change to, standard coherence protocols supported by most high-performance memory buses today. The next two chapters compare the performance of CNIs against alternative ULNI designs.[1]

This chapter begins by describing *cachable device registers* (CDRs) and *cachable queues* (CQs). A CDR is a coherent, cache block used by a processor to communicate information to or from a CNI device (Section 3.1). A CQ generalizes this concept into a contiguous region of coherent, cache blocks (Section 3.2). Because CDRs and CQs can be

1. Available partially in [85, 89].

cached in processor and CNI caches, they require a *home*, which is an I/O device or memory module that services requests and accepts writebacks for CDR and CQ blocks (Section 3.3).

Section 3.4 describes a concise taxonomy of the CNI design space exposed by CDRs, CQs, and their homes. Section 3.5 shows that with adequate support some CNIs can also be attached to the I/O bus. Section 3.6 discusses how to multiprogram CNIs. Section 3.7 briefly outlines how CNIs can be interfaced with standard networks. Section 3.8 discusses related work. Finally, Section 3.9 summarizes the techniques described in this chapter. [1]

3.1 Cachable Device Register (CDR)

A cachable device register (CDR) is a coherent, cachable block of memory shared between a processor and a CNI device. Reinhardt, et al. [101] first proposed *cachable control registers* (CCRs) to communicate status information from a special-purpose hardware device to a processor. This thesis extends their work to use coherence to efficiently communicate control information and data both to and from an ULNI device. I call such registers CDRs. This section examines how a CDR works (Section 3.1.1), its advantages (Section 3.1.2), and its disadvantages (Section 3.1.3).

3.1.1 Basic CDR operation

A CDR is a coherent, cachable memory block shared between a processor and a CNI device. Like the Thinking Machine CM-5’s memory-mapped NI registers, CDRs are memory-mapped into a user’s virtual space. However, unlike the CM-5 NI registers, CDRs are cachable. I assume that each process will negotiate its own CDR from a CNI. However, if multiple processes require access to the same CDR, they must impose some kind of synchronization (e.g., implemented via shared-memory primitives between the multiple processes).

1. Thanks to David Wood for suggesting the general approach of caching network interface registers.

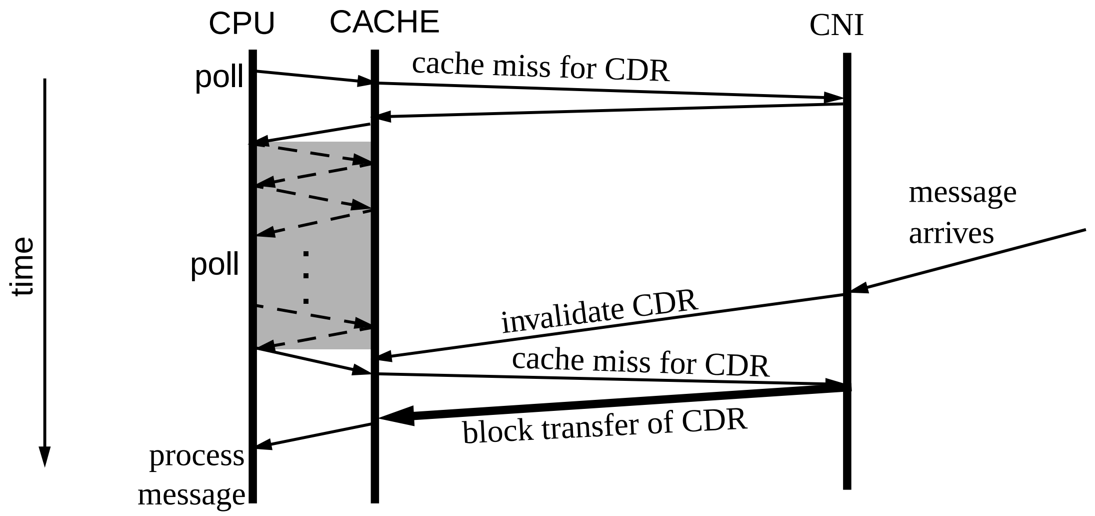


Figure 3-1. CDR Transfer Example. This figure shows CDR transfers between the CPU, CPU's cache, and a CNI, assuming write-allocate caches kept consistent by a MOESI write-invalidate coherence protocol [123]. Initially, the CPU polls a CDR to check the presence of a message. Assume this incurs a cache miss. This cache miss is satisfied by the CNI (instead of main memory), which indicates the absence of any message in the CNI. The CPU's subsequent polls to the CDR block (shaded region) is satisfied directly from its cache. Finally, a message arrives, which prompts the CNI to invalidate the CDR block in the CPU's cache following the standard coherence protocol supported by the memory bus. The subsequent cache miss for the CDR block brings in the new CNI status and the first few words of the message in one block.

A CNI sends information to a processor—i.e., to initiate a request or update status—by writing to the *virtual address* of the CDR. However, the CNI must first obtain write permission to the CDR block. Such write permission can be obtained via the underlying coherence protocol. That is, the processor will incur a cache miss for the CDR block. The CNI will observe the *physical address* of the CDR block on the bus—just like a regular cache miss—and respond appropriately.

A processor may receive information from a CNI by polling a CDR block (via its virtual address). Unlike existing polling schemes, the CDR block is cachable, so in the common case of unchanging information, the processor's unsuccessful polls normally hit in the local cache.[1] Bus traffic only occurs when a device updates the information. Figure 3-1 illustrates how a CDR is transferred between a CPU, the CPU's cache, and

1. Cache conflicts can cause replacements, which affect performance but not correctness.

the CNI. Because a CDR consists of a whole cache block and status information is typically less than a few words, part of a message can be communicated between the CPU and the CNI in a single bus transaction, amortizing the fixed overheads across multiple words.

3.1.2 Advantages of a CDR

A CDR improves performance in four ways. First, in the common case of unchanging information, e.g., polling, a CDR removes unnecessary bus traffic because repeated accesses hit in the cache. With conventional uncached device registers, each poll by the processor must go across the bus to the device. In a symmetric multiprocessing node, repeated polls to uncached device registers can consume bus bandwidth, which could be used by other processors in the node.

Second, when changes do occur, a CDR uses the underlying coherence protocol to transfer messages a full cache block (e.g., 32-128 bytes) at a time. However, for smaller amounts of data, e.g., a 4-byte word, CDRs are less efficient. For most processors, fetching a single word from an uncached device register takes roughly the same time as from a CDR; this is because the CNI responds with the requested word first which is then bypassed to the processor. However, the CDR still has higher overhead since it will displace another block from the cache, potentially causing a later miss. CDRs do even less well for small transfers to a device. Because most modern processors have store buffers, a single uncached store is more efficient than transferring that word via a CDR. For most processors and buses the breakeven point typically occurs at two or three double words. Hence, my CNI implementations (Section 4.1) with non-speculative processors use uncached stores to transfer single words of control information from the processor to the device. However, with a dynamically-scheduled, speculative processor (not evaluated in this thesis), uncached loads and stores can stop the processor's out-of-order and speculative accesses. Consequently, in such processors it may still be advantageous to transfer even small amounts of data via CDRs to avoid messing up the processor's out-of-order and speculation engine.

Third, a CDR can transfer information both from the device to the processor as well as from the processor to the device in a logically symmetric way. Thus, a CNI device can poll a CDR and directly read messages from the processor's cache using the standard coherence mechanisms. CNIs can further optimize these polls through a technique called *virtual polling*. Because a CNI device observes the coherence protocol directly, it knows when the processor requests write permissions to the block. Hence, it need not poll periodically, but can read the block back soon after the processor requests permission. The CNI device can provide a system programmable back-off interval to reduce the likelihood of "stealing" the block back before the processor competes its writes to the CDR. This technique, called virtual polling, is useful for processors that cannot efficiently "push" data out of their caches. For processors (e.g., PowerPC [132]) that do support user-level cache flush instructions, a CDR can be directly flushed out of the cache.

Fourth, processors can efficiently communicate control information, e.g., interrupt masks, to an NI through CDRs. Changing control information, such as masking NI interrupts, can be expensive in modern processors this may require an uncached store, a write buffer flush, and a trip through the operating system. To avoid these costs, Stodolsky, et al. [121] proposed the *optimistic interrupt protection* scheme based on the assumption that interrupts are rare events for short critical sections. A derivative of this scheme is implemented in the CM-5 version of the Wisconsin Blizzard system [104]. In Blizzard, the software assigns a global processor register to hold a software copy of the hardware NI interrupt mask register. When a processor enters or leaves a critical section, instead of turning interrupts off before and on after the critical section, it simply changes the NI interrupt level in the global register only. If an NI interrupt does occur during the critical section, the operating system checks the global register and sets a status bit in the register. The processor checks this status bit when it exits the critical section and executes the corresponding interrupt handler. [1]

CDRs provide an alternative and perhaps a more elegant solution to the optimistic interrupt protection scheme. In a CDR-based optimistic interrupt protection scheme, the interrupt mask register is put in a CDR and cached by the processor. In the common case of no interrupts within a short critical section, the processor simply writes to the CDR that contains the interrupt mask register in its cache before and after the critical section. When an interrupt does occur, the NI reads and invalidates the CDR in the processor's cache and sets a separate status bit in the same CDR. On exiting the critical section the processor checks the CDR status bit, incurs a cache miss for it, reads the new status when the cache miss is satisfied by the CNI, and executes the interrupt handler. This scheme has two advantages over Stodolsky, et al.'s scheme. First, it removes the operating system from the communication path between the processor and CNI. Second, it does not reserve a register from the global register pool, which can be a precious resource (particularly for x86 machines).

3.1.3 Disadvantages of a CDR

The main disadvantage of CDRs is that they require some method for reuse. This makes them work less well when a processor reads multiple cache blocks of the same message or different messages using the same CDR. For example, after the processor has read the first block of a message, it may want to read the second block using the same CDR. Conventional device registers often solve this problem using an implicit *clear-on-read* semantics, where the register is cleared after an uncached read. For example, the CM-5 network interface treats the read of the hardware receive queue as an implicit "pop" operation. Clear-on-read works because processors guarantee the atomicity of individual load instructions; that is, the value returned by the device is guaranteed to be written to a register.

Clear-on-read does not work well for CDRs, since most processors do not provide the same atomicity guarantees for cache blocks. The load that causes the cache miss should be

1. If the processor checks the status bit first and then resets the interrupt mask when it exits a critical section, then it may lose an interrupt. Resetting the interrupt mask first and then checking status bit guarantees that no interrupt is lost.

Steps	Processor
One	Uncached store to NI
Two	Flush store buffer
Three	Poll uncached NI register

Table 3.1: Steps in the three-cycle handshake

atomic (to close the “window of vulnerability” [63]); however, there are no guarantees for the remaining words in the block. Before subsequent loads complete, a cache conflict (e.g., resulting from an interrupt) could replace the block. With clear-on-read semantics, the remainder of the data in the CDR would be lost forever. Consequently, for CDRs to work correctly, they must have an explicit clear operation by the receiver to enable reuse of the block.

Under a MOESI protocol even this clear operation requires a slow three-cycle handshake between the processor and CNI to make sure that the processor sees new data when it re-reads the CDR (Table 3.1). In the first step of this handshake, the processor issues an explicit clear operation by performing an uncached store to a traditional device register. In the second step, the processor must ensure that the CNI has seen the clear request. Since most modern processors employ store buffers, this step may incur additional stalls while a memory barrier instruction flushes the store out to the bus. When the CNI observes the explicit clear operation, it invalidates the CDR by arbitrating for and acquiring the memory bus. The third step of the handshake is for the processor to ensure that the invalidation has completed. It does this by reading, potentially repeatedly, a traditional uncached device status register.[1] Consequently, while CDRs efficiently transfer a single block of information, they perform much less well for multiple blocks. CQs, described next, solve this problem by amortizing the cost of the handshake over several cache blocks.

1. A somewhat more efficient handshake is possible if the processor provides a user-accessible cache-invalidate operation. Issue clear operation, flush store buffer, and invalidate cache entry.

3.2 Cachable Queues (CQs)

Cachable Queues (CQs) generalize the concept of CDRs from one coherent memory block to a contiguous region of coherent memory blocks managed as a queue. CQs are a general mechanism that can be used to communicate messages between two processor caches or a processor cache and a device cache. For the purpose of this thesis, I will design CQs to communicate messages between a CNI and a processor cache. I expect that each user process will negotiate at least two CQs—one to send messages and the other to receive messages from the CNI—with the CNI.

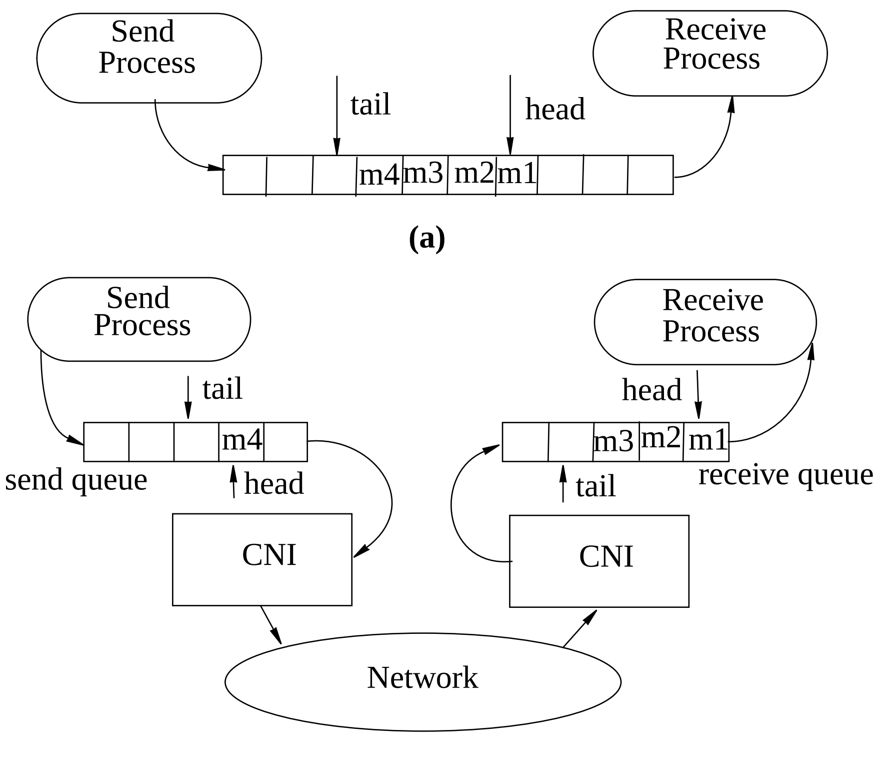
A key advantage of CQs is that they simplify the reuse handshake and amortize its overhead over the entire queue of blocks. Liu and Culler [70] used cachable queues to communicate small messages and control information between the compute processor and message processor in the Intel Paragon.

This section focuses on how a single user processor can use CQs to communicate messages directly from a network interface device. I first describe the basic queue operation (Section 3.2.1) and then introduce five important performance optimizations (Section 3.2.2). Section 3.6 will discuss how CQs can be used by multiple processes.

3.2.1 Basic CQ Operation

Cachable queues follow the familiar enqueue-dequeue abstraction and employ the usual array implementation, illustrated in Figure a. The head pointer (`head`) identifies the next queue entry to be dequeued, and the tail pointer (`tail`) identifies the next free queue entry. The queue is empty if `head` and `tail` are equal, and full if `tail` is one entry less than `head` (modulo queue size). If there is a single sender and single receiver for this queue, the case I consider in this thesis, then no locking is required since only the sender updates `tail` and only the receiver updates `head`[1].

1. Memory barrier operations may be necessary to preserve ordering under weaker memory models.



(b)

Figure 3-2. Cachable Queues. (a) Local Cachable Queue (b) Remote Cachable Queue. m1, m2, m3, and m4 denote valid messages sent by the Send Process to the Receive Process.

A processor sends a message by simply enqueuing it in the appropriate out-bound message queue. In particular, it first checks for overflow, writes the message to the next free queue location, and increments `tail`. A processor receives a message by checking for an empty queue and reading the queue entry pointed to by the `head`. The message remains in the queue until the receiver explicitly increments `head`. The `head` and `tail` reside in separate cache blocks.

Because CQs are simply memory, they have the property that the message sender and receiver have the same interface abstraction whether the other end is local or remote. A local CQ, illustrated in Figure 3-2a, is simply a conventional circular queue between two processors. A remote CQ consists of two local CQs, each between a processor and CNI device, as illustrated in Figure 3-2b. The `head` and `tail` are also managed as cachable memory. A CNI that uses CQs simply acts like another processor manipulating the queue.

The head and tail pointers of the CQs are a much simpler way to manage reuse than the complex handshake required by CDRs. If there is room in the CQ, then the tail entry can be reused; if the CQ is non-empty, then the head entry is valid. However, even though no locking is required to access the `head` and `tail`, a straight-forward implementation induces significant communication between sender and receiver. This occurs because the sender must check (i.e., read) the `head`, to detect a full queue, and the receiver must check `tail`, to detect an empty queue. Because the queue pointers are kept in coherent memory, cache blocks may ping-pong with each check. This overhead can be greatly reduced using five techniques described in the next subsection.

3.2.2 CQ Optimizations

This section describes five optimizations—*lazy pointer*, *message valid bit*, *sense reverse*, *empty entry removal*, and *intra-message prefetch*—that can greatly reduce the overhead to access CQ entries. I describe the CQ optimizations assuming fixed-size CQ entries. However, the message valid bit optimization, and consequently sense reverse that depend on the message valid bit, require slight modifications to work with variable-size queue entries. I will point out the changes in appropriate places. [1]

To explain these optimizations I examine the steps involved in enqueuing and dequeueing messages to and from a CQ. Each CQ has a sender and a receiver. For the send queue (Figure 3-2), the processor is the sender and CNI the receiver. For the receive queue (Figure 3-2), the CNI is the sender and the processor is the receiver. For a regular circular

1. Optimizations described in this section will also work for a CQ set up as a linked list of fixed-size or variable-size entries. The only difference would be that the sender and receiver would have to do an extra pointer dereference to get to the next CQ entry.

queue the pseudo-code for enqueuing and dequeueing are as follows:

```
if (tail ⊕1 != head)
{
   enqueue message at tail
   tail = tail ⊕1
}
```

Regular enqueue

```
if (tail != head)
{
   dequeue message from head
   head = head ⊕1
}
```

Regular dequeue

⊕ denotes addition modulo CQ size. In the pseudo-code above and the ones to follow, I assume that if the queue is full, then the sender will either stall or return an error condition on an enqueue operation. The receiver behaves similarly on a dequeue operation.

As the pseudo-code above shows, on every enqueue operation, the sender must read the `head` shared between the sender and the receiver. Unfortunately, in the worst case, this may cause a cache miss for the `head` on every enqueue operation, because the receiver may concurrently dequeue the queue entries and increment the `head`.

Lazy Pointer. The *lazy pointer* optimization allows the sender to avoid reading the shared `head` on every enqueue operation. Lazy pointers exploit the observation that the sender need not know exactly how much room is left in the queue, but only whether there is enough room. The sender maintains a (potentially stale) copy of the `head`, `shadow_head`, which it checks before each send. `Shadow_head` is conservative, so if it indicates there is enough room, then there is. Only when `shadow_head` indicates a full queue does the sender read `head` and update `shadow_head`. Thus, the pseudo-code for enqueue with the lazy pointer optimization looks as follows:

```
if (tail ⊕1 != shadow_head || tail ⊕1 != (shadow_head = head))
{
   enqueue message at tail
   tail = tail ⊕ 1
}
```

Enqueue with lazy pointer

Note that the above pseudo-code assumes C language semantics for the “||” operator in which the second condition is not evaluated if the first condition is true. Additionally, if the first condition is false, `shadow_head` acquires the value of `head`, even if the second condition is false.

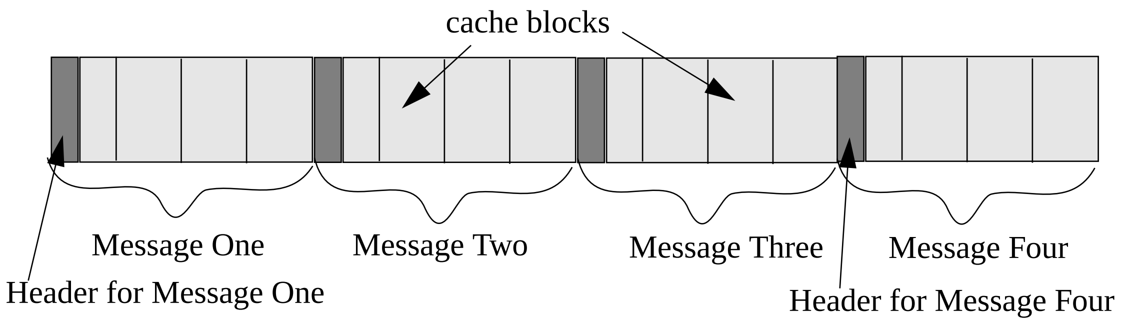


Figure 3-3. A Cachable Queue (CQ) with four messages.

The lazy pointer optimization works well when, on average, there exists several empty CQ entries. For example, if the CQ is no more than half full on average, then the sender needs to check `head`—and incur a cache miss—only twice each time around the array.

Message Valid Bit. For `tail`, however, a *message valid bit* provides a better optimization than a lazy tail pointer. Tail pointer works poorly if messages arrive one at a time. In contrast, a message valid bit allows the receiver to detect message arrivals without ever checking `tail`, thereby eliminating the need for a shared `tail`.

The message valid bit can be stored either as a single bit or a separate word in the message header. For my implementation of CQs, the message follows the message header (Figure 3-3). For all subsequent pseudo code, I will assume that the message valid bit is the first word of the message header.

The sender sets the valid bit when it writes a new message to a CQ entry. This may involve invalidating the receiver's cached copy of the CQ block containing the message (and the message valid bit). When the receiver reads the same CQ location again, this invalidation will force the receiver to obtain the new cache block from the sender via a cache miss. The receiver will first check if the message valid bit is set to ensure the presence of a new message. Subsequently, when the receiver is done reading the message, it must reset the valid bit before it advances `head`. No explicit synchronization (e.g., a lock) is required to update the valid bit because the sender updates the bit only when it is not set, while the receive updates it when it is set.

The pseudo-code for enqueue and dequeue with the message valid bit would be as follows:

```
if (tail ⊕ 1 != shadow_head ||
    tail ⊕ 1 != (shadow_head =
                 head))
{
   enqueue message at tail
   *tail = VALIDBIT
   tail = tail ⊕ 1
}
```

Enqueue with message valid bit

```
if (*head == VALIDBIT)
{
   dequeue message from head
   *head = ~VALIDBIT;
   head = head ⊕ 1
}
```

Dequeue with message valid bit

Message valid bits are not new. The *T-NG network interface [22] supports uncached message valid bits. Liu and Culler [70] used cached message valid bits in their Paragon Active Message implementation.[1] The Scalable Coherent Interface [116] optionally supports a primitive called QOLB (queue on lock bit) directly in the coherence protocol. This lock bit (per cache block) could be used as a cached message valid bit. A message valid bit differs from QOLB's lock bit because a message valid bit can be implemented on top of a coherence protocol; it does not have to be integrated with a coherence protocol, as is done for QOLB.

Cached message valid bits, as offered by CQs, improve performance in two ways. First, in the absence of any message processor polls to a message valid bit will hit in the processor cache. Second, when a message does arrive and a processor incurs a cache miss for the message valid bit, it obtains the first cache block worth of the message (minus the valid bit

1. An alternative optimization to message valid bits is possible for CQs for which the processor is the receiver and the CNI is the sender. If the processor can selectively invalidate cache blocks, then after reading a message, it can invalidate the CQ cache blocks that contained the message. Subsequent accesses to these CQ blocks will result in cache misses. The CNI can respond to such cache misses only when a valid message is available. However, this scheme has three disadvantages. First, few processors support such invalidate instructions. Second, the processor will be blocked on the cache miss. Third, memory buses that do not support split-transactions would require the CNI to repeatedly send negative acknowledgments to the cache miss request and the processor cache to repeatedly send the cache miss request until it is satisfied. This would unnecessarily waste bus bandwidth. Even some split-transaction buses may time out if the response does not arrive within a certain interval [135].

or word) from the CNI. This amortizes the overhead of cache miss incurred due to a message valid bit.

The message valid bit optimization requires slight changes for variable-size CQ entries. The receiver for a CQ must be able to locate the message valid bit for the next message in a CQ to check for the presence of a new message. The location of a new message for fixed-size CQ entries is fixed. For example, for a CQ that starts at address X and has entries of size 256 bytes, message valid bits will be located at X, X+256, X+512, and so on. These addresses are fixed across all passes through the CQ. Consequently, the message valid corresponding to a particular location, say X+256, remains the same in the next pass through the CQ, unless a new message has been written by the sender into that CQ location. For variable-size CQ entries, however, a new message starts where a previous message ends. Consequently, message valid bits can be located at different offsets in different passes through the CQ because messages themselves can be of different sizes. This also implies that the message valid bits from a previous pass can be overwritten by the body of new messages in the current pass through the CQ. Therefore, when the receiver checks the next word following a freshly-read message, it may incorrectly see the body of an old message, instead of a message valid bit. To prevent this the sender must also write the message valid bit of the next message when it writes a new message (and its valid bit) into the CQ.

Sense reverse. Clearing the message valid bit requires the receiver to write the queue entry; thus under a MOESI protocol, the receiver becomes owner of the queue entry’s cache block, rather than simply having a shared copy. This normally requires an additional bus transaction. This transaction (and clearing of the valid bit) can be avoided using a technique called *sense reverse*. The key idea is to alternate the encoding of the valid bit on each pass through the queue. Valid is encoded as 1 on odd passes, and encoded as 0 on even passes. The sender and receiver both have an additional state bit, stored in the same cache blocks as their respective pointers, indicating the sense of their current pass.

Below I show the pseudo-code for the simple case where the valid bit is stored in a separate word in the header. The sender first checks if the CQ has space and then writes the message followed by its current sense as the message valid bit. The receiver compares its current sense to the valid bit in the message, with a match indicating a valid message.

```
if (tail ⊕ 1 != shadow_head ||
    tail ⊕ 1 != (shadow_head =
                 head))
{
   enqueue message at tail
   *tail = sender’s sense;
   tail = tail ⊕ 1
   if (tail == 0)
   {
      sender’s sense =
         sender’s sense xor 1
   }
}
```

Enqueue with sense reverse

```
if (*head == receiver’s sense)
{
   dequeue message from head
   head = head ⊕ 1
   if (head == 0)
   {
      receiver’s sense =
         receiver’s sense xor 1
   }
}
```

Dequeue with sense reverse

Sense reverse has been previously used for barriers [79] and asynchronous logic, but to the best of my knowledge has never been used for messaging.

The sense reverse optimization requires changes similar to the message valid bit optimizations for variable-size CQ entries. On each enqueue operation, the sender must first write the sense bit of the next message in the CQ array, before it writes the current message.

Empty entry removal. Valid bits (and, hence, the sense bits) provide a fourth opportunity for optimization. I call this *empty entry removal*. Conventional circular queue implementations often use an empty queue entry to help distinguish between queue full and queue empty conditions. For example, the queue empty condition would be (`tail` == `head`), while the queue full condition would be (`tail` ⊕ 1 == `head`). However, given the valid bits, the empty queue entry can be removed. With valid bits the queue empty and full conditions would translate into (`tail` == `head` && valid bit not set) and (`tail` == `head` && valid bit set) respectively. With sense bits, the queue empty and full conditions are somewhat subtle. The conditions are (`tail` == `head` && sender's sense == receiver's sense) and (`tail` == `head` && sender's sense != receiver's sense) respectively. Updates to the receiver's sense and `head` must, however, be atomic, because they jointly comprise the receiver's view of the CQ. Thus, the pseudo-code for CQ enqueue and dequeue with all the optimizations look as follows:

```
if (tail != shadow_head ||
    tail != (shadow_head = head)||
    sender's sense == receiver's
                        sense)
{
   enqueue message at tail
   *tail = sender's sense;
   tail = tail $\oplus$ 1
   if (tail == 0)
   {
      sender's sense =
         sender's sense xor 1;
   }
}
```

Enqueue with empty entry removal

```
if (*head == receiver's sense)
{
   dequeue message from head
   tmp_head = head $\oplus$ 1
   if (tmp_head == 0)
   {
      tmp_sense =
         receiver's sense xor 1
      atomically update head
         and receiver's sense
         with tmp_head and
         tmp_sense resp.
   }
   else
      head = tmp_head
}
```

Dequeue with empty entry removal

If a processor is the receiver and `head` and receiver's sense are allocated in separate words, then atomic updates to `head` and receiver's sense can be achieved by allocating `head` and receiver's sense in a single double-word variable and performing a single double-word store to this variable. This ensures atomicity because processors guarantee atomicity of individual store operations. If a hardware finite state machine mimics the receiver (e.g., in a CNI), then it can guarantee atomicity by simultaneously updating both `head` and the sense.

The empty entry removal optimization is worthwhile only for small CQs. Large CQs may afford to waste an entry because it consumes a small fraction of the total CQ space. Also, this optimization does not help much for variable-size CQ entries because `head` and

CQ state at sender	**CQ state at receiver**
Shared `head`	Shared `head`
Private `shadow_head`	Private sense bit
Private `tail`	Translations (Section 3.6.2)
Private sense bit	Process identifier (Section 3.6.4)
Translations (Section 3.6.2)	▨
Process identifier (Section 3.6.4)	▨

Table 3.2: CQ state at sender and receiver

`tail` would be incremented in 4-byte word intervals. Consequently, using one 4-byte word to distinguish between CQ empty and full conditions is not that wasteful.

Intra-message prefetch. The fifth optimization—*intra-message prefetch*—is a variant of virtual polling (Section 3.1.2) that minimizes the number of bus transactions on the critical path for the send CQs (i.e., a CQ where the processor is the sender and a CNI is the receiver). Specifically, under the bus's write-invalidation based MOESI protocol, the processor must generate an invalidation signal to acquire ownership of a cache block before it can write to it. Since our CQs are filled in FIFO order, an invalidation signal for all blocks other than the first block of a multi-block message implies that the processor is done writing the previous cache block. When the CNI device detects an invalidation signal it issues a coherent read on the previous cache block of the same message. Thus part of the message is transferred to the CNI cache before the processor has completed writing all the cache blocks of the message.

Combining all five CQ optimizations—lazy pointer, message valid bit, sense reverse, empty entry removal, and intra-message prefetch—minimizes bus traffic and space required by CQs. Table 3.2 shows that a CQ incurs very little overhead at the sender and receiver. Table 3.3 summarizes the benefits of the five CQ optimizations. In the common case, these optimizations reduce both the number of invalidation and read misses from

Optimization	**Benefit**
Lazy pointer	sender does not read head pointer (in common case) to write a new message into the CQ
Message valid bit	receiver does not read tail pointer to read a new message from CQ
Sense reverse	receiver does not write the CQ
Empty entry removal	saves space in CQ
Intra-message prefetch	receiver overlaps the consumption of previous blocks a message with generation of new blocks

Table 3.3: Summary of CQ optimizations.

three (for `head`, `tail`, and message) to one (for message). This result holds for a write-invalidation based MOESI protocol and cache-block sized messages. The reduction can be even greater for update protocols and messages larger than a cache block.

3.3 Home

In most computer systems, all legal physical addresses map to a *home* device or memory module. If a block is cachable, for example, then the home is where data are written on cache replacement. Should the home for CDRs or CQ entries be at the CNI, as with a regular device register, or in main memory?

Since CDRs are each a single block and most devices will employ only a few, the logical choice is to provide the home within the device itself. This can also simplify the implementation for some memory buses, because the device may not have to implement all cases in the coherence protocol [97]. For example, cache replacement for CDR blocks need not be implemented because the CNI itself is the home for the CDR block.

CQs, on the other hand, will benefit from being large. For example, Brewer, et al., have demonstrated that remote queues can significantly improve performance by preventing contention on the network fabric [16]. If the CQ's home is main memory—a less precious resource than hardware FIFOs—then its capacity can be very large (e.g. megabytes). Large queues help simplify protocol deadlock avoidance and flow control, at least for moderate-scale parallel machines. Having the CQ home in memory also helps tolerate unreliable network fabrics, since messages need not be removed from the send queue until delivery is confirmed.

Mapping the home of CQs to main memory raises several operating system and performance issues. I discuss these in the next two subsections.

3.3.1 Operating System Issues

To place the CQ home in main memory, we must address three operating system issues. First, a CNI needs a translation scheme to translate the CQ virtual addresses to physical addresses in main memory. If the operating system allocates CQ pages contiguously, then CNIs can use a simple base-and-bounds virtual-to-physical address translation. If the operating system cannot guarantee this, then a more complicated translation mechanism may be necessary. Section 3.6.2 discusses several alternate translation mechanisms.

Second, a CNI must ensure that CQ pages always reside in main memory, or be prepared to fetch them from the swap device. For the CNI implementations in this thesis, I assume that CQ pages are “pinned,” so that the operating system does not attempt to page them out. Alternatively, more flexible schemes are possible (see Section 3.6.1).

Finally, there must be some mechanism for the rare case in which even the large amount of memory allocated for a CQ fills up. Three options exist. The first and the simplest option is to block the sender; however, this may lead to deadlock. Second, as proposed for MIT Fugu [72], the CNI device can interrupt the processor, causing it to allocate free virtual memory frames and drain the CQ. Third, the CQs themselves could grow dynamically. An easy way to achieve this would be organize the CQs as a linked-list of fixed-size buffers. If a CNI finds that the receive CQ is full when it tries to write new messages to the CQ, it can interrupt the OS. The OS can make an upcall to the user process, causing it to allocate virtual memory for the buffers. After obtaining the virtual memory frames for the buffers, the OS must allocate backing physical memory and insert these buffers in the CQ. A similar approach could be taken for the send CQ. This approach has the disadvantages of requiring a pointer dereference for every message access and a more elaborate scheme to manage the translations of the buffers.

3.3.2 Performance Issues

Treating CNI memory as a CQ cache requires the CNI to handle cache replacements of modified cache blocks from the CNI cache. Modified cache blocks result in a CNI's cache from messages that arrive from the network. For a FIFO-style CQ, flushes are unnecessary in two cases:

- A message may have been consumed by a processor (i.e. a *dead* message), but is flushed to main memory because cache blocks it resides in are marked modified.
- New messages arriving from the network continuously replace fresh messages from the CNI cache to main memory. This can happen when the CNI cache overflows due to a burst of messages arriving from the network. This may cause the processor to pick up all messages from main memory, instead of the smaller and faster CNI cache. The negative impact of this problem can be reduced if new messages arriving from the network bypass the cache and write them directly to main memory.

Below I describe two optimizations—*dead message elimination* and *cache bypass*—that help remove these two unnecessary cache flushes.

Dead Message Elimination. A *dead message* is a message received by a CNI and already consumed by the processor. Dead messages from the receiving CNI's cache need not be flushed to main memory. Figure 3-4 shows that dead messages can be determined easily by comparing the `head` and `tail` of the CQ.

The dead message elimination optimization changes the contract between the CNI and a processor. After a processor increments `head` of the receive CQ, the CNI does not guar-

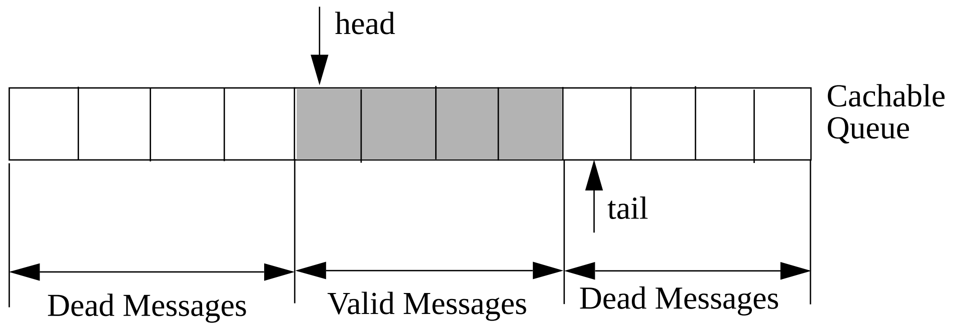


Figure 3-4. Dead message elimination. A simple comparison of the head and tail pointer shows messages that are already consumed by the receiver.

antee the validity of data for the dead message in the previous CQ entry. However, the CNI must still guarantee the validity of the sense bit in that CQ entry to prevent the processor from incorrectly interpreting the presence of a valid message in the CQ entry. Consequently, even with dead message elimination CNIs must still flush the sense bit of a dead message to main memory.[1] This may be accomplished using an uncached store from the CNI to main memory.

The lazy pointer optimization (Section 3.2.2) creates a problem in determining dead messages in the receiving CNI's cache. This is because the lazy pointer optimization, when successful, allows the `shadow_head` and `head` to drift apart. Consequently, messages may be dead, even though indicated otherwise by the `shadow_head`. To solve this problem, I force the CNI to update the `shadow_head` after every N cache block flushes from the CNI. I call N the dead message elimination threshold. This allows the CNI to have a more precise view of dead messages in the CQ and allows a better elimination of cache flushes for dead messages.

My experiments in Chapter 4 suggest that one is a reasonable value for N. N = 1 suggests that not only do messages arrive in batches, which allows `shadow_head` to improve performance, but also messages are consumed by the processor in batches. Con-

1. A writeback buffer may help improve the performance of writeback of sense bits to main memory. However, my experiments (not shown in this thesis) indicate that such a writeback buffer has negligible effect on the performance of CNIs.

sequently, if the CNI encounters one dead message, the probability of encountering consecutive dead messages is high.

Of the seven macrobenchmarks evaluated in this thesis (Table 4.3), dead message elimination only helps spsolve significantly. For this benchmark a dead message elimination threshold of one improves the performance of $CNI_{32}Q_m$ by 12%. The improvement is greater for CNI_iQ_m with $i > 32$. See Section 3.4 for an explanation of the CNI taxonomy.

Cache Bypass. If messages arrive from the network at a CNI faster than the processor can consume them, then the CNI cache will overflow. This may cause the processor to pick up all messages from main memory. However, because CNI caches are smaller, we expect them to be faster. Therefore, allowing the processor to read messages from the CNI cache, instead of main memory, should improve performance. Consequently, writing new messages to main memory directly and bypassing a full CNI cache should improve performance.

I use a heuristic to determine if the CNI should bypass its cache when a new message arrives from the network. In this heuristic each CNI cache block is augmented with one state bit, which indicates if the cache block has been directly read by the processor at least once from the CNI's cache. Before writing a block from a new message (arriving from the network) into the CNI cache, the CNI must check if the state bit of the old cache block indicates whether the processor has read the block at least once. If so, then the CNI can safely replace the old block, write a block from the new message into the same location in the cache, and reset the state bit.[1] However, if the state bit is not set, implying that the processor has not read it even once, then the CNI bypasses its cache and writes the block directly to main memory.

1. Of course, cache replacements in the processor cache can cause the processor to request the block again. Nevertheless, I expect such cache replacements from the processor's cache to be rare, and, therefore, replacing the corresponding block from the CNI's cache is worthwhile.

Chapter 4 shows that this optimization improves performance of two macrobenchmarks—em3d and spsolve (Table 4.3)—by 4% and 8% respectively.

3.4 CNI Taxonomy

This section proposes a taxonomy of network interfaces (NIs). I use the NI queue structure as the main component to enumerate a taxonomy of network interfaces. NI queues are the primary carriers of messages between a processor and its NI. A processor sends messages to the NI through the *send queue* and receives messages from the NI through the *receive queue*. For our taxonomy of CNIs, I assume that both the NI queues have the same structure.

This taxonomy is modeled after Agarwal et al.'s classification of directory protocols [4]. I use the notation NI_iX for traditional FIFO-based NIs and CNI_iX for coherent network interfaces that cache the NI queues. The subscript *i* denotes the size of the NI FIFO queue exposed to the processor. The default unit of *i* is memory/cache blocks, but can also be specified in 4-byte words by adding the suffix 'w'. The placeholder X could either be empty, Q, or Q_m. X empty represents the simple case where a network interface exposes only part or whole of one network message. For CNIs a network message is exposed via CDRs. CDR reuse is managed by the explicit handshake described in Section 3.1.3. X = Q represents the more complex case where the exposed portion of the NI queue is managed as a memory queue with explicit head and tail pointers. X = Q_m denotes that the home of the explicit memory-based NI queues are in main memory. The absence of an '*m*' implies that the device serves as the home for the NI queues.

For example, *i* = 2w denotes that only two words of the FIFO queue is exposed. A processor reads words of a message from an NI_{2w} network interface by issuing uncached loads to a fixed address, which repeatedly pops the NI_{2w} FIFO. Similarly, a DMA-based NI that reads or writes up to 64 words of a message at a time via a DMA engine can be specified as NI_{64w} (see Section 5.3). In contrast, for some network interfaces, such as

$NI_{128}Q$, the processor must explicitly increment and dereference a head pointer to read different words of a message from different addresses of the queue.

Several existing NIs can be classified with this taxonomy. The Thinking Machines' CM-5 [124] NI is NI_{2w} since it exposes two words of a message to the receiver. Similarly, the Alewife [2] NI is NI_{16w} [62]. The network interface in *T-NG [22], which devotes 8 KB for an NI queue and consists of 64-byte cache blocks, is $NI_{128}Q$. The *T-Jr NI [53] can be classified as CNI_0Q_m because it does not have a cache (hence '0'). I call the *T-Jr NI a CNI, even though it does not have a cache, because it allocates its queues in main memory. If the *T-NI were on the memory bus, it would have to issue coherence signals to keep these queues coherent between main memory and the processor caches.

Chapter 4 compares the performance of NI_{2w}, CNI_4, $CNI_{32}Q$, $CNI_{512}Q$, and $CNI_{32}Q_m$ with two microbenchmarks and seven macrobenchmarks.

3.5 CNIs on I/O buses

Although coherent memory buses allow processors and CNIs to coherently cache CQs, the same may not be possible with CNIs on all I/O buses (Table 3.4). Two key mechanisms are necessary in the interconnect between the processor, the CNI, and the home to coherently cache CQs in a processor or CNI cache. First, if a cache contains the most recent copy of a CQ block, it must be able to intercept a coherent read request for the block and prevent the home from responding. Second, a cache must be able to invalidate (or update) stale copies of CQ blocks residing in other caches.

The absence of the first mechanism makes it difficult for I/O bus CNIs to coherently cache CQs whose home is in main memory. Main memory resides on the memory bus, while the I/O bus is usually connected to the memory bus through an I/O bridge. Because I/O buses are usually slower than memory buses and the I/O bridge introduces additional delay on the memory bus to I/O bus path, it is difficult for I/O bus devices to intercept a memory bus coherent read request and prevent main memory from responding in a timely

Home	**Caching CQs In**	**Non-Coherent I/O**	**Coherent I/O**	**Coherent I/O + I/O Bridge Invalidation Support**
Main Memory	CNI Cache	No	No	No
	Processor Cache	Slow	Yes	Yes
CNI	Processor Cache	Slow	Slow	Yes

Table 3.4: Coherent Network Interfaces on I/O buses

fashion. Memory bus CNI caches, on the other hand, directly observe the memory bus coherence protocol and therefore can intercept a coherent read request, inhibit memory from responding (through the memory inhibit signal on the memory bus), and respond with the most recent copy of the CQ block. Although I/O bus CNIs cannot cache CQ blocks that reside in main memory, they could write messages directly to main memory, just like regular network interfaces that DMA messages to main memory.

The absence of the second mechanism—i.e. the ability to invalidate or update stale CQ blocks in other caches—in today's I/O buses makes it difficult for processor caches to coherently cache data from an I/O bus CNI device when the home is in main memory. This is because when a CNI updates main memory the processor cache can still contain stale data. Invalidations can, however, be synthesized at the software level or at the I/O bridge. At the software level, a processor could explicitly flush its entire cache (or the CQ blocks selectively, if selective invalidations are allowed) before a CNI writes new data to CQs in main memory. However, this solution—adopted in today's systems that support only *non-coherent I/O*—is slow because this requires a cache flush and an explicit handshake between a CNI and processor before the CNI can write new messages to main memory. Alternatively, many systems support *coherent I/O* by adding functionality at the I/O bridge. On a I/O device write data to main memory, the I/O bridge invalidates all stale copies of data residing in memory bus caches. The same mechanism could be used by CNIs to allow processors to cache CQ blocks.

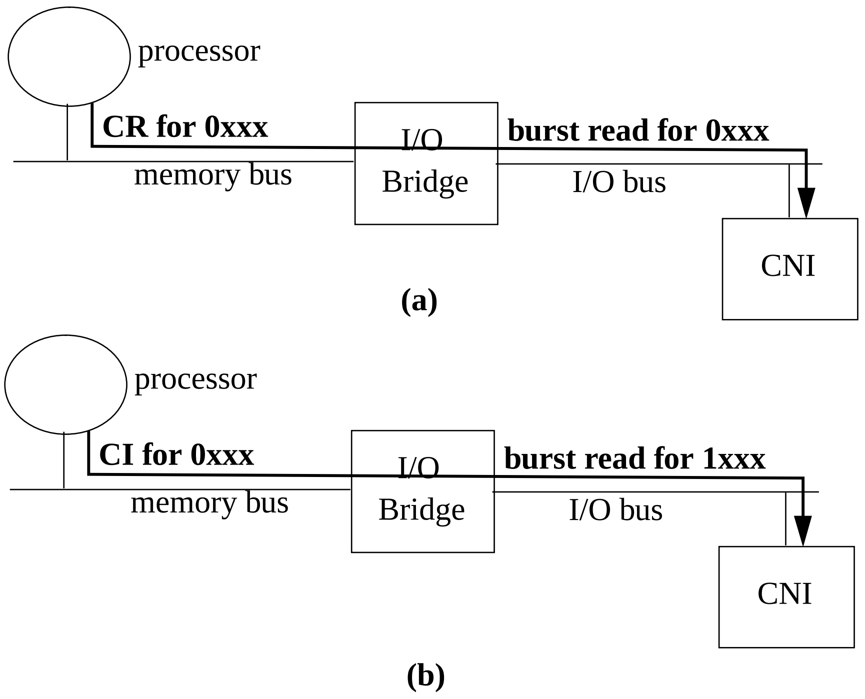


Figure 3-5. Illustration of shadow address with a CNI on the I/O bus. (a) a processor read miss (CR = coherent read) for the address 0xxx is translated by the bridge to a burst read for 0xxx on the I/O bus. (b) shows that a CI (coherent invalidate) signal for 0xxx is translated to a burst read for the shadow address 1xxx. The CNI interprets a read for 1xxx as a invalidation signal for 0xxx.

I propose a third alternative in which the I/O bridge synthesizes invalidation signals on an I/O bus, such the Sun SBus or PCI[1], using a technique called the *shadow address space*. Recall invalidation signals are necessary to allow a processor to avoid having stale copies of CQ blocks in the CNI's cache or memory when the processor writes new messages to the CQ. Similarly, invalidation signals help a CNI to avoid having stale copies of CQ blocks in the processor cache when the CNI writes new messages arriving from the network into the CQ.

The shadow address space technique has been used before to communicate special signals from a processor to an I/O device [11, 47], but not in an I/O bridge. In this technique, the I/O bridge creates a shadow space for the regular I/O space by some invertible function

1. PCI supports only two coherent transactions: memory read line and memory write and invalidate. The invalidate command. Consequently, we need to fake an invalidation signal even on PCI.

such as flipping a bit. Thus, if 0xxx represents an I/O space physical address, 1xxx will represent its shadow physical address. Reads from the processor to 0xxx will proceed normally, but reads to 1xxx will be interpreted by the I/O device as a special control operation, which in our case is an invalidation on the physical address 0xxx (Figure 3-5). Thus, when the I/O bridge observes an invalidation signal for address 0xxx on the memory bus, it will convert it to a read signal on the address 1xxx. Conversely, when the I/O bridge observes an I/O bus read signal on 1xxx, it will convert it to a memory bus invalidation signal on the address 0xxx. These enable an I/O device to observe all memory bus invalidation signals for CQ blocks and send invalidation signals for CQ blocks to memory bus caches. In this thesis, for all experiments with I/O bus CNIs, I assume that the I/O bridge supports the shadow address space technique. I will refer to an I/O bus augmented with I/O bridge invalidation support as a *coherent I/O bus*.

If the home of the CQs is in the CNI device itself, then the first mechanism is not necessary. However, if processors cache CQ blocks, then a CNI must invalidate the processor’s stale CQ blocks when new messages arrive from the network. The invalidation schemes discussed in the preceding paragraph applies in this situation too.

There are other alternative designs for CNIs on I/O buses. For example, although I/O buses may not allow main memory requests to be intercepted by I/O bus devices, they could allow I/O bus devices to intercept requests to memory residing on the I/O bus itself. This would enable us to place the home in memory on the I/O bus. Another option is to design the CNI to a standard I/O bus (e.g., PCI) specification, but provide special bridges (such as the personality interfaces on the SGI Power Challenge I/O bus [42]) through which the CNI can talk to a variety of memory buses (Section 2.2).

3.6 Multiprogramming CNIs

Emerging server applications, traditional parallel programs, and the advent of symmetric multiprocessing nodes (SMPs) make it critical to multiprogram network interfaces (NIs) of high-end servers and parallel machine nodes. World-wide web, database, and network computer query servers constitute new classes of emerging server applications that must rapidly respond to incoming message requests. High-end server machines that run multiple such server applications must be multiprogrammed to allow maximum overlap of CPU and I/O operations from these servers. Throughput-oriented production runs of traditional parallel applications, such as scientific codes, may not necessarily require a multiprogrammed parallel machine. Nevertheless, parallel machines must be multiprogrammed to allow fast response times for program development workloads. Finally, the advent of SMPs as high-end servers and parallel machine nodes allow the opportunity to run multiple processes in parallel within the SMP node itself. Multiprogramming processor and memory for communication-intensive server and parallel applications is, however, almost useless unless we multiprogram the NI itself.

I see four key design challenges to multiprogramming an NI. These are:

- Ensuring protected access to the NI (Section 3.6.1),
- Providing the NI with physical addresses of message data buffers (Section 3.6.2),
- Allowing multiple processes to simultaneously access the NI (Section 3.6.3), and
- Detecting generation of new messages from different processes (Section 3.6.4).

Below I discuss each of these design challenges. For each design problem I first examine the alternatives that exist in today's commercial and research NIs and explain which solutions are most suitable for CNIs. Section 3.6.5 discusses an example send and receive datapath through a multiprogrammed CNI_iQ_m device.

3.6.1 Ensuring Protected Access to the NI

Protected user access to status, control, and data registers of an NI isolates user processes from one another and prevents one process from clobbering another process's messages. There are two standard ways to ensure protected, user access to NI registers. The first, and the traditional method, is to access the NI through the operating system (OS). User applications send messages to and receive messages from the OS through the conventional system call interface. In Unix for example, users can send and receive messages to and from the network via the socket interface. This scheme ensures protection because only the OS directly talks to the NI. Unfortunately, as discussed in Section 2.8, routing messages through the operating system can incur high latency (tens of microseconds), because of the need to cross a protection boundary. [1]

In the second approach followed by CNIs, NI registers are memory-mapped directly into user virtual space, which allows direct, protected, and rapid access to the NI registers. In this approach conventional virtual memory hardware guarantees protection on a per page basis. To the best of my knowledge, the Thinking Machines' CM-5 NI was the first to adopt this mechanism. Several recent NI designs, such as the Princeton SHRIMP NI [12], Mitsubishi DART [96], etc. have adopted a similar strategy for protection. All CNI designs adopt this approach. CDRs and CQs are memory-mapped into user virtual space; the virtual memory hardware guarantees protection for these data structures shared between the NI and a user process.

Memory-mapping NI registers to user space introduces a complexity for systems with NIs that directly read and write messages to and from the processor's memory space. Protection will be violated if the operating system remaps a physical page to another process or swaps a physical page to disk when an NI is reading from or writing to that page. DMA-based NIs and CNIs fall in this category of NIs. For DMA-based NIs, this situation can arise while DMA is in progress. For CNIs this situation arises during cache block writebacks from a CNI cache to main memory.

Researchers have proposed two solutions to this problem of protection violation caused by page remapping and swapping. CNIs can use either. First, all pages that an NI reads from or writes to can be pre-allocated and pinned to main memory. This ensures that all physical pages to or from which an NI transfers data are never remapped or swapped. The

1. Even extensible operating systems, such as SPIN [10] or VINO [108], cannot eliminate the overhead of trapping into the operating system. Extensible operating systems can, however, reduce the overhead of heavy-weight protocols, such as UDP/IP, by customizing them for specific applications.

Arizona Application Device Channels [35] and the Cornell U-Net architecture [127] have taken this approach. This scheme is simple to implement and requires no or very little change to commodity operating systems and NIs. However, this limits the amount of user virtual memory space that an NI can address because only limited amounts of physical memory can be pinned throughout the execution of a program. This prevents NIs from directly depositing data into any place in user virtual space.

Second, instead of pinning all pages that the NI can potentially write data to or read data from throughout the entire duration of a program, the OS and the NI can cooperate to temporarily "pin" only those pages that are involved in data transfer between the processor's memory space and the NI (e.g., SHRIMP I[11], U-Net/MM [133]).[1] This allows the NI to address the entire user virtual space. However, the OS must be careful not to remap pages that have data transfers in progress. Both SHRIMP and U-Net/MM disallow the OS remapping such pages. In SHRIMP the OS is disallowed from remapping pages that are actively involved in message transfers. The SHRIMP NI hardware posts addresses of such pages in its registers. The OS simply reads these registers before remapping or swapping any page; if a page's address is currently in an NI register, then the OS avoids remapping or swapping that page until the NI clears that address from the registers. U-Net/MM is more coarse-grained. It disallows any further remapping of a page that has been mapped by a translation cache (Section 3.6.2) located in the NI. The OS can only remap pages whose translations have been replaced from or do not appear in the NI's translation cache.

CNIs with caches impose an additional requirement on the OS and NI. Before a page remap or page swap, modified CNI cache blocks corresponding to a page must be flushed to main memory. The OS can use the same technique it uses to flush cache blocks from processor caches. The OS running on a processor simply reads the entire page into the processor cache and then writes the page to main memory. The same protection mecha-

1. Alternatively, the OS itself could dynamically pin the pages before initiating data transfer between main memory and the NI. But this has higher overhead because the OS must be invoked on every message send and reception to update the page table data structures that typically contain this information.

nisms described earlier in this subsection can be used to prevent the CNI to re-read the same blocks. Alternatively, the CNI can coordinate with the OS to flush all cache blocks directly from the CNI cache to main memory.

3.6.2 Providing the NI with Physical Addresses of Message Data

NIs that read data from or write data to processor caches, NI caches, or main memory must know the physical address of the message data buffers. NIs that transfer data to or from main memory using DMA provide a classic example of this kind of NIs. CNIs face a similar challenge because CNIs must be prepared to retrieve data from processor caches and write data to main memory or CNI caches themselves. Coherent memory buses simplify the solution for NIs because NIs need not worry about where exactly the data is located; the data can either be in main memory or processor caches. The NI simply specifies the physical address of the message data buffer and the memory bus returns an up-to-date copy of the message data to the NI. As discussed in Section 3.5, this is known as coherent I/O.

There are two standard ways in which an NI can acquire physical addresses of data buffers: either the processor can provide the NI with the physical address of the data buffers just before the message send or receive or the user application can directly send the virtual address of the message data buffers to the NI device. In the latter case, the NI must have a mechanism to translate the user virtual address to the physical address in the system. CNIs can adopt either method; nevertheless, for performance reasons, I would recommend the second approach.

In the first and the more traditional method, the OS supplies NI devices with physical addresses because user applications do not have access to and cannot be trusted with physical addresses of NI devices or main memory. This requires OS intervention on every message send and receive. Recently, the Princeton UDMA mechanism relaxed this constraint by showing how DMA can be initiated directly from user-level through the *shadow address space* technique [11] using simple uncached loads and stores to the NI device.[1]

The shadow address space technique allows a user process to transfer physical addresses for the source and destination of DMA to the NI without violating protection. This strategy works well for processor-initiated sends and processor-initiated message receives. However, for asynchronous message reception such as the one offered by the Active Message model, this may hinder performance because the NI cannot directly deposit data into user space without processor intervention. Alternatively, as in the Princeton SHRIMP approach [12], the sending NI can negotiate ahead of time the physical addresses to which messages would be deposited in the receiving node. On a message send, the sending NI appends to the message the appropriate physical address in the receiving node. This approach allows asynchronous message reception without processor intervention. The drawback of this approach is that any local change in translation (e.g., page remap or swap) requires the system to reflect this change globally, which can be an expensive operation. This can be avoided by pinning the destination pages for the entire duration of a parallel program.

In the second method, the NI holds the user virtual-to-physical translations so that it can directly interpret user space virtual addresses provided by the user. There are several alternatives for this—from storing the entire page table in the NI [107], to caching the translations in NI data structures, either in software [47] or in hardware [96]. There are two problems associated with caching the translations in the NI: how to fill the NI translation buffer and how to avoid stale copies of translations when the operating system has remapped a page or swapped a page to disk. The NI translation buffer can be filled in two ways. First, on a translation buffer miss, the NI can interrupt the operating system, which can insert the requested translation into the NI translation buffer. Second, translations can be inserted into the NI translation buffer directly by the user through the shadow address space technique [11, 47, 101] using simple uncached loads or stores to the ULNI device.

1. See Chapter 5 for details on User-Level DMA

The translation table in the NI must be updated if the operating system either remaps a page or swaps a page out to disk. If the operating system decides to remap a page, it can simply invalidate the corresponding translation in the NI. The operating system may have to wait for an access is in progress from the NI on that particular page to complete. Alternatively, the operating system can ensure that pages mapped by the NI's TLB are always pinned in memory and never reclaimed unless the NI explicitly gives them up [133].

Page swaps can be handled in two ways. First, all physical pages may be pinned in main memory for the duration of the program. For long-running programs, this solution may be infeasible. Second, the operating system can again invalidate the translation in the NI translation buffer when it swaps a page out to disk. When the NI accesses the page again, it will incur an NI translation buffer miss and request the host processor to remap the page. The host processor will swap the page back into physical memory and reinsert the translation into the NI translation buffer. [1]

3.6.3 Allowing Multiple Processes to Simultaneously Access the NI

The advent of SMPs as high-end servers and parallel computer nodes has made it critical to allow multiple processes to access the NI registers simultaneously. Traditionally, NIs have provided only one set of registers (status, control, and data) that are accessed and controlled only by the OS. This solution works for SMPs if all message sends and receives are routed through the OS. However, User-Level NIs (ULNIs) do not have this privilege because they must allow user applications to directly access the NI registers. ULNIs such the TMC CM-5 NI [124] and the Princeton SHRIMP NI [13] suffer from a similar problem; although these NIs allow multiple processes to access the NI registers, they do not allow more than one user process to simultaneously access the NI registers.

A common approach to allowing multiple processes to access the NI registers simultaneously is to provide multiple sets of NI registers and memory-map a different set into dif-

1. For more detailed discussions on protection and address translation issues in the network interface, please see Heinlein, et al. [47], Blumrich, et al. [11], and Schoinas and Hill [105].

ferent user virtual spaces. The Arizona Application Device Channels (ADCs) [35], the Cornell U-Net [127], and Fujitsu AP-Net [111] have all taken this approach. The key interface abstraction for these approaches is a *memory-based queue* (Section 2.7). CNIs take a similar approach; additionally, CNI_iQ_m's ($i > 0$), which are CNIs with caches whose home is in main memory, provide greater flexibility in multiprogramming. Processes communicate with CNIs through CQs, which are memory-based queues with the additional property that these queues can be cached in the processor and CNI caches.

Memory-based queues are allocated by either dividing up the fixed amount of memory in the NI among multiple queues or using main memory as a repository for these queues. The first method limits the number of queues that can be allocated because NI memory is usually a small and precious resource and for each process the queues must be allocated in multiples of page sizes (e.g., 4 - 16 kilobytes) because the standard virtual memory hardware guarantees protection only at the page granularity. Consequently, AP-Net supports only two user-level queues, while the ADC implementation described in [35] supports 16 user-level queues. Alternatively, these NIs can support a larger number of queues if the OS is prepared to save and restore the queues and queue state when the number of queues allocated exceeds the total amount of memory on the NI. However, since message queues can be quite large (tens of kilobytes), saving and restoring these queues and queue state can be an expensive operation. The CNI_iQ designs—CNIs with no caches—suffer from this problem. The second method, adopted by the U-Net architecture alleviates this problem somewhat by allocating the queues in main memory and queue state in the NI memory. Since the queues are in main memory and managed through regular virtual memory hardware and software, they need not be saved and restored. However, the disadvantage of this method is that messages cannot be transferred directly between the NI and processor; all messages must be routed through main memory.

CNI_iQ_m's ($i > 0$) simplify the multiprogramming problem because they allow a system to support a large number of queues with very little memory in the NI. The key observation is that they separate the logical allocation of CQs from their physical location. Logically, CQs are allocated in virtual address space of each process at page granularity. Physically, however, they can be located in the CNI caches, processor caches, or main memory, as need be. Like the ADC implementation, they allow messages to be directly transferred between the NI and the processor; like U-Net the number of CQs supported is only restricted by the number and amount of CQ state that can be allocated in the NI memory. Since the amount of CQ state per CQ is typically small (Table 3.2)—that is, tens of bytes—the number of CQs that can be supported is very large.

The number of CQs supported by a CNI can be made even larger, if need be so, by making main memory the home of the CQ state and caching the CQ state in the CNI. The Berkeley Active Messages implementation on a Network of Workstations running the Solaris operating system and connected with Myricom Myrinet switches [74], caches state specific to network connections (called endpoints) in the NI memory. However, the Myrinet host interface does not interact with the host system through coherent operations. Consequently, the Berkeley implementation must keep such state coherent in software. Alternatively, because CNIs interact with the host system through coherent memory operations, the CQ state itself can be allocated in coherent memory and treated exactly like CQs themselves.

3.6.4 Detecting the Generation of New Messages by Different Processes

CNIs must have efficient mechanisms to detect the generation of new messages by different processes, both during message send and reception. Message detection has two parts: detection of message creation and association of a message with a specific process.

On the send side, in almost all current NI designs, processes inform the NI of the creation of a new message through an uncached signalling store to the NI. This approach is appealing because of its simplicity. Nevertheless, as discussed in Chapter 2, uncached loads and stores can be slow and often stop the speculation engine of modern microprocessors. In the absence of uncached signalling stores, CNIs must be prepared to monitor the state of each CQ to detect the presence of a new message—either by monitoring changes in the tail pointer or by detecting changes in the message valid bits. Monitoring a large number of CQs can be an expensive operation because the CNI has to check the status of each CQ for new messages. Virtual polling (Section 3.1.2) can be a viable alternative here, particularly in the presence of a large number of CQs. A CNI device can be mostly passive; it needs to poll a CQ only when an invalidate coherence signal appears at the CNI's bus interface. Associating a message with a specific process is simple for CQ-based CNIs because different processes will write messages to different physical addresses.

On the receive side, a CNI logically detects the presence of a new message either by polling the external network interface or through an external network interface interrupt. The actual state machine can be implemented either in firmware or as a finite-state machine. To associate an incoming message with a receiving process, the message provides the CNI with some identifier. Examples of such identifiers include global process identifiers and CQ identifiers. However, such identifiers must be added to the message by the sending CNI, and not the user, because CNIs cannot trust users to provide correct identifiers. Such identifiers can be obtained from the operating system and maintained along with the CQ state (Table 3.2).

3.6.5 Multiprogrammed CNI_iQ_m Datapath

Figure 3-6 shows the logical path followed by a message through a multiprogrammed CNI_iQ_m device. Figure 3-6a shows the logical path followed on a message send. The message arrives from coherent memory bus. The CNI_iQ_m device examines the queue identifier and looks up the queue state in the queue state table. It translates the virtual queue pointer addresses and any other virtual address specified in the message to the corresponding physical addresses through the TLB (translation lookaside buffer). In parallel the device fetches the corresponding cache blocks from the cache to which the message must be written. When the translation completes and the message is written to the cache, the device signals the external network interface of the presence of a new message. When the net-

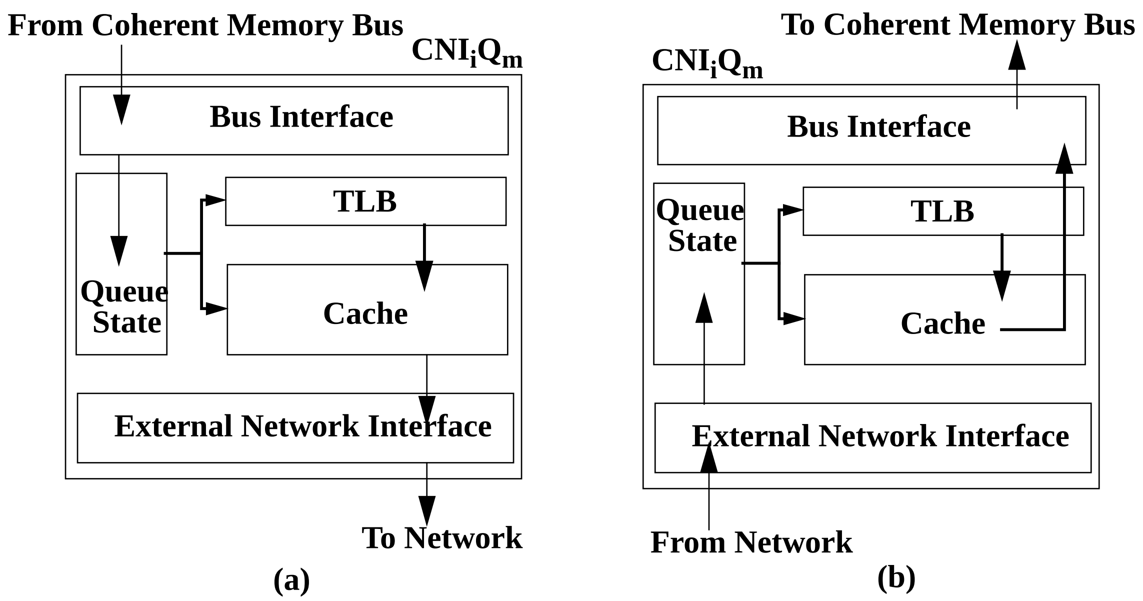


Figure 3-6. Logical datapath through multiprogrammed CNI_iQ_m device. This figure shows the logical path through followed by a message for a message send (a) and message receive (b) for a multiprogrammed CNI_iQ_m device.

work is ready to receive messages, the external network interface reads the message from the CNI_iQ_m cache and injects it into the network.

Figure 3-6b shows the logical receive datapath through a CNI_iQ_m device. When the message comes in from the network and through the external network interface, the CNI-$_iQ_m$ device examines the queue identifier from the message, looks up the queue state table, extracts the virtual addresses from the queue state and the incoming message, consults the translation table for the corresponding physical addresses and fetches the corresponding cache block in parallel, and finally inserts the message into the CNI_iQ_m cache. At some later time, the corresponding user process will read the message from the CNI_iQ_m cache.

For both send and receive the CNI_iQ_m device may have to postpone writing new messages into the cache, if the corresponding cache blocks in the device cache are marked modified and must be flushed to main memory.

3.7 Interfacing CNIs with Standard Networks

Although CNIs are a novel class of network interfaces, they can be interfaced with standard networks in three ways:

- A user process can communicate with an *agent* using CQs. This agent is a user process that runs on the same processor (or different processor in a Symmetric Multiprocessing node) and communicates with the nodes standard network interface on behalf of user process.
- A user process can communicate directly with a standard network interface using CQs. For example, the Myrinet host interface [15] could allow CQs if it could interface to a memory bus and reflect coherence signals to its on-board microprocessor.
- Another alternative is to standardize the interface between the internal and external network interfaces (Section 2.2), so that each memory bus vendor can provide its own internal interface. Such an internal interface could implement CNI techniques to optimize data transfers between a processor cache and the network interface.

3.8 Related Work

Coherent Network Interfaces differ from most previous work on program-controlled network I/O in three important respects. First, unlike other NIs, CNIs interact with processor caches and main memory primarily through the node's coherence protocol. Second, CNIs separate the logical and physical locations of NI registers and queues allowing processors to cache them like memory. Third, CNIs provide a uniform memory-based interface for both local and remote communication. Table 3.5 compares network interfaces of different machines with respect to these three issues. The Thinking Machines' CM-5 [124], the Wisconsin Typhoon [100], the Stanford FLASH [64], and the Meiko CS2 [78] multiprocessors provide high latency uncached access to their NIs on the memory bus. Since both Typhoon and FLASH have a coherent cache in their network interfaces, they could both support CQs. The Meiko CS2 network interface supports the memory bus's coherence protocol, but does not contain a cache. The MIT Alewife [2] and FUGU [72] machines

Network Interface	Coherence	Caching	Uniform Interface
CNI	Yes	Yes	Memory Interface
TMC CM-5 [124]	No	No	No
Typhoon [100]	Possible	Possible	Possible
FLASH [64]	Possible	Possible	Possible
Meiko CS2 [78]	Possible	No	Possible
Alewife [2]	No	No	No
FUGU [72]	No	No	No
StarT-NG [22]	No	Maybe	No
AP1000 [110]	No	Sender	No
T-Zero [103]	Partial	Partial	No
SHRIMP [12]	Yes	Write Through	No
DI Multicomputer[23]	No	No	Network Interface

Table 3.5: Comparison of CNI with other network interfaces

provide uncached access to their NIs under control of a custom CMMU unit. The StarT-NG NI [22] is not coherent because it is a slave device on the non-coherent L2 coprocessor interface. StarT-NG NI queues can be cached in the L1 cache, but the processor must explicitly self-invalidate or flush stale copies of the NI queues. Wisconsin T-Zero [103] caches device registers, but not queues, and only uses them to send information from the device to the processor. AP1000 [110] directly DMA's messages from the processor's cache to the NI, but does not receive messages directly into the cache. Princeton SHRIMP's memory bus NI [12] allows coherent caching on the processor, but requires processors to use the higher traffic write-through mode. The DI multicomputer's on-chip NI [23] neither supports coherence nor allows its registers or queues to be cached. The processor chip interfaces with the rest of the system through the NI. Unlike other machines, the DI-multicomputer supports a uniform message-based interface for both memory and the network, whereas CNI uses the same *memory* interface for both memory and network.

Unlike many other NIs, my *implementation* of CNIs does not require changes to an SMP board or other standard components. Yet they enable processors and network interfaces to communicate through the cachable memory accesses, for which most processors and buses are optimized. Henry and Joerg [50] and Dally, et al. [34] advocate changes to a processor's registers. MIT Alewife [2] and Fugu [72] rely on a custom cache controller. MIT StarT-NG [22] requires a co-processor interface at the same level as the L2 cache. AP1000 [110] requires integrated cache and DMA controllers. Stanford FLASH [64, 48] uses a custom memory controller with an embedded processor. Other efforts, such as the TMC CM-5 or SHRIMP, use standards components, but settle for lower performance by using loads and stores to either uncachable or write-through memory, instead of using the full functionality of write-back caches.

Five efforts that appear very similar to this work are FLASH messaging [47], UDMA/SHRIMP-II [11], Remote Queues [16], Cray T3E messaging [107], and SCI/QOLB [116]. CNIs differ from FLASH, because they do not require a processor core in the network interface, they allow commands to use cachable loads and stores, and they can notify the receiving process without an interrupt.

CNIs differ from the UDMA/SHRIMP-II, because CNIs use the same mechanisms when the destination is local and remote (whereas SHRIMP-II's UDMA does not handle local memory to local memory copies), CNIs use only virtual addresses (where SHRIMP-II requires that the sender knows the receiver's physical addresses), they allow device registers to use writeback caching, and they focus on fine-grain user-to-user communication in which the receiving process may be notified without an interrupt.

CNIs differ from Remote Queues by being at a lower-level of abstraction. Remote Queues provide a communication model similar to Active Messages [128], except extracting a message from the network and invoking the receive handler can be decoupled. Implementing Remote Queues with CNIs is straightforward and offers advantages over CM-5, Intel Paragon, MIT Alewife, and Cray T3D network interfaces. CNIs support cachable device registers for low-overhead polling (unlike the others), allow network buffers to gracefully overflow to memory (unlike the CM-5), and do not require a second processor

(Paragon), custom cache controller (Alewife), or hardware support for globally shared memory (T3D).

Like Remote Queues, the Cray T3E queues are at a higher level of abstraction. CQs differ from T3E queues because they are allocated by operating system, whereas T3E queues are allocated by the user itself. CQs are communication structures between a local processor and a local CNI. Consequently, there exist separate send and receive CQs, which together make a remote CQ. In contrast, in T3E the NI is unaware of any such queue structure. CQs can automatically wrap around for reuse without user intervention, whereas the T3E queues require the user to explicitly manage the queues. Finally, because a CNI is aware of a CQ’s structure optimizations, such as intra-message prefetch (Section 3.2.2), are possible.

A CNI differs from the SCI coherence protocol, because a CNI is *not* a coherence protocol itself; instead, a CNI interfaces to a coherence protocol. A CNI can be designed to a memory interconnect that supports most standard coherence protocols, including SCI’s coherence protocol. However, QOLB (queue on lock bit)—a feature optionally supported by SCI—can help improve the performance of a CNI. QOLB allows efficient implementation of the producer-consumer sharing pattern by allowing direct transfer of a single cache block from the producer to the consumer. By queueing the consumer’s request for a producer’s cache block, QOLB prevents the consumer from repeatedly stealing the cache block that the producer is writing to. Because a processor cache and a CNI constitute a producer-consumer pair, CNIs can effectively use the QOLB primitive of SCI. However, for send CQs in which the CNI is the consumer, the virtual polling technique can potentially achieve the same effect as QOLB.

3.9 Summary

This chapter explored a novel class of network interfaces called *coherent network interfaces (CNIs)* that use snooping cache coherence to improve communication performance between processors and network interfaces. CNIs use two mechanisms that CNIs use to communicate with processors. A *cachable device register (CDR)* allows information to be exchanged in whole cache blocks and permits efficient polling where cache misses (and bus transfers) occur only when status changes. *Cachable queues (CQ)* reduce re-use overhead by using array of cachable, coherent blocks managed as a circular queue and (optionally) optimized with lazy pointers, message valid bits, sense-reverse, empty slot removal, and intra-message prefetch. Because CDRs and CQs can be cached in processor and CNI caches, they require a *home*, which is an I/O device or memory module that services requests and accepts writebacks for CDR and CQ blocks. Either the CNI itself or main memory can serve as the home for CDRs and CQs. For CQs whose home is in main memory and that are cached in CNI caches, I developed two mechanisms—*dead message elimination* and *cache bypass*—to minimize cache flushes from the CNI cache to main memory. Based on CDRs, CQs, and home for CDRs and CQs, I developed a CNI taxonomy that captures a wide range of traditional NIs and CNIs. Finally, I examined how to interface CNIs to I/O buses and the operating system support needed to multiprogram CNIs.

Chapter 4

An Evaluation of Coherent Network Interfaces

Coherent Network Interfaces (CNIs) constitute a new class of user-level network interfaces (ULNI) that interact with the processor via the node’s coherence protocol. The optimal CNI design effectively exploits all eight opportunities for optimization described in Chapter 2. The previous chapter examined specific techniques for interfacing CNIs with a node’s coherence protocol. It also exposed a spectrum of alternative CNI designs, each with different performance characteristics.

This chapter evaluates the performance of four CNIs from the CNI design space and compares them against a more traditional ULNI, like the TMC CM-5 NI. Section 4.1 describes the implementation of these five NIs I evaluated in this chapter. Section 4.2 describes the evaluation methodology I used in this chapter and the rest of the thesis. Section 4.3 and Section 4.4 present detailed results from this evaluation with two microbenchmarks and seven macrobenchmarks, respectively. Finally, Section 4.6 presents my conclusions.

The next chapter focuses on data transfer and buffering—two of the eight optimizations described in Chapter 2—and compares and evaluates CNI techniques for data transfer and buffering against alternative ULNI designs.

4.1 Network Interfaces Simulated

This section describes the implementation of five network interfaces (NIs)—NI_{2w} (Section 4.1.1), CNI_4 (Section 4.1.2), $CNI_{32}Q$ (Section 4.1.3), $CNI_{512}Q$ (Section 4.1.3), and $CNI_{32}Q_m$ (Section 4.1.4)—for 256-byte network messages. These NIs are summarized in Table 4.1. In Section 4.3 and Section 4.4, I will evaluate and compare the performance of these NIs.

4.1.1 NI_{2w} Implementation

NI_{2w} is a conventional network interface modeled after the Thinking Machines CM-5 NI. Messages are sent by first checking an uncachable status register, to ensure there is room to inject the message, then the message is written to an uncachable device register backed by a hardware queue.[1] Similarly, receives check an uncached status register to see if a message is available, then read the message from an uncachable device register. Because all accesses to the NI queues are non-cachable, and two four-byte words of the message are exposed, I classify this device as NI_{2w}.

1. In the TMC CM-5, a user process (or, software message library) first writes all words of a message to the NI, and then checks the send status register to make sure that the NI has accepted the message. If not, it writes the message again to the NI. Avoiding the status register check before sending the message is an optimization that saves an uncached load on the critical path of the message send. I avoid this optimization because of three reasons. First, I assume a network message size of 256 bytes. Consequently, the relative overhead of status check is lower than the CM-5, which has a network message size of only 20 bytes. Second, this makes my comparison uniform with the CNIs, which cannot blindly write messages without checking the status register. This is because the message queues reside in memory shared between the processor and the CNI. Third, writing a message directly to the NI without knowing if the NI will accept the message requires the messaging software to buffer these messages in certain cases. This can incur extra overhead and reduce overall performance.

NI/CNI	Exposed Queue Size	Queue Pointers	Home
NI_{2w}	2 words		
CNI_4	4 cache blocks		device
$CNI_{32}Q$	32 cache blocks	explicit	device
$CNI_{512}Q$	512 cache blocks	explicit	device
$CNI_{32}Q_m$	32 cache blocks	explicit	main memory

Table 4.1: Summary of Network Interface Devices.

4.1.2 CNI_4 Implementation

CNI_4 extends NI_{2w}—my baseline NI device—by using four 64-byte CDRs to expose a 256-byte network message. CNI_4 exploits the memory bus's block transfer capability to move a message between the processor and the device. However, the status and control registers are uncached. When a message arrives at the CNI_4 device, the device checks if the CDRs are free. If so, it writes the message to the CDRs and sets a status register indicating the presence of a message. The processor checks this status register, finds a new message, and incurs cache misses for the CDRs to read the message. Finally, after the processor is done reading the message, it issues an uncached store to the CNI_4 device that signals that the CDRs can be reused. Unfortunately, these checks introduce a three-cycle handshake and reduces the effective bandwidth between the processor and the NI (see Section 3.1.3).

4.1.3 $CNI_{32}Q$ and $CNI_{512}Q$ Implementations

$CNI_{32}Q$ and $CNI_{512}Q$ amortize the cost of CNI_4's three-cycle handshake by employing CQs (Section 3.2) for message data and regular memory for control and status information (head and tail pointers). $CNI_{32}Q$ and $CNI_{512}Q$ cache up to 32 and 512 blocks, respectively. The memory that backs up the caches resides on the devices themselves. The larger capacity of $CNI_{512}Q$ reduces the number of flow control stalls, increasing performance for applications with many messages in flight.

Sending messages to a CNI_iQ device involves three steps: checking for space in the CQ, writing the message, and incrementing the tail pointer. The send is further optimized by sending a message ready signal to the CNI device through an uncached store. As discussed in Section 3.1.2, with a non-speculative, in-order processor, which I use for all my evaluations in this thesis, uncached stores are more efficient than cache block operations for small control operations. Hence, for the send queue, the CNI device does not use virtual polling. Instead, the CNI_iQ uses the message ready signal to keep a count of pending messages. This count is incremented on each message ready signal and decremented when the device injects a message into the network. As long as this counter is greater than zero, the CNI_iQ device pulls messages out of the processor cache (unless the blocks have already been flushed to their home in the device) and increments the head pointer. On the receive side, the processor polls the head of the queue, reads the message when valid, then increments the head pointer. Both sender and receiver toggle their sense bits when they wraparound the end of the CQ.

4.1.4 $CNI_{32}Q_m$ Implementation

The $CNI_{32}Q_m$ device caches up to 32 cache blocks for each CQ on the network interface device, and overflows to main memory as necessary. The total size of the memory-based queue is 512 cache/memory blocks. Having main memory as home for the CQ simplifies software flow control. Specifically, for the other NIs, whenever the sender cannot inject a message it must explicitly extract any incoming messages and buffer them in memory [16]. Conversely, $CNI_{32}Q_m$ does this buffering automatically when the CNI cache cannot contain all the messages.

4.2 Simulation Methodology

This section describes the Wisconsin Wind Tunnel II simulator (Section 4.2.1), the simulation parameters (Section 4.2.2), and the macrobenchmarks (Section 4.2.3) I used to evaluate the five NIs described in Section 4.1. Section 4.3 and Section 4.4 present results from the evaluation.

4.2.1 Wisconsin Wind Tunnel II Simulator

I use the Wisconsin Wind Tunnel II (WWT II) [92][1] simulator for all my simulations in this thesis. WWT II is a fast and portable simulator for parallel architectures. I developed WWT II jointly with Babak Falsafi, Mike Litzkow, and Steve Reinhardt. WWT II inherits many features of the original Wisconsin Wind Tunnel (WWT) [99, 91], including distributed, discrete-event simulation techniques [40], direct-execution [27], and accurate calculation of a simulated architecture's execution time via executable editing [65]. However, unlike WWT, which only runs on the TMC CM-5, we designed WWT II to be easily portable. Consequently, WWT II runs on several uniprocessor and multiprocessor SPARC platforms, including sparcstations, SUN SMP enterprise servers, and a cluster of sparcstations connected via Myrinet Myricom switches. This ability to run WWT II on several platforms offers simulation cycles that is orders of magnitude greater than that available from WWT.

4.2.2 Simulation Parameters

All my simulations use system parameters specified in Table 4.2, unless specified otherwise. I use an aggressive one-GHz, dual-issue HyperSPARC-like processor. Although my simulations do not model a dynamically-scheduled processor, which is likely to dominate in the future, I believe that both my quantitative comparisons and qualitative trends can be extrapolated to these processors. This is because my primary focus is on *relative* performance of different NIs using the same base processor model, and not on the absolute performance of a particular processor architecture. Additionally, my relative performance results are conservative for CNIs. This is because out-of-order and speculative processors hide memory latencies better than in-order processors. Because CNIs are memory-based NIs, they can make better use of such latency-hiding techniques compared to NI_{2w}.

1. WWT II's release information is available from http://www.cs.wisc.edu/~wwt/wwt2.

System Parameters	
Number of parallel machine nodes	16
Processor speed	1 GHz
Cache block size	64 bytes
Cache size	one megabyte
Cache associativity	direct-mapped
Main memory access time	120 ns
Memory bus coherence protocol	MOESI
Memory bus width	256 bits
Memory bus clock frequency	250 MHz
I/O bus width	64 bits
I/O bus clock frequency	125 MHz
Network message size	256 bytes
Network latency	40 ns
NI memory access time	60 ns

Table 4.2: System parameters for simulated system. All simulations in this chapter use the above parameters, unless specified otherwise. The NI memory access time is 60 ns for all NIs except CNI_iQ. Because CNI_iQ's home resides in the CNI itself, the CNI memory must be large to support a large degree of multiprogramming. Hence, I expect it to be built with commodity DRAM with access time characteristics similar to main memory (i.e., 120 ns). Note that a CNI_iQ_m device interface with a 120ns access time would perform similar to a CNI_iQ device when the CNI_iQ_m cache overflows rarely. Thus, sensitivity to this parameter on CNI_iQ_m's performance can be interpreted indirectly from my results in Section 4.4.1.

All my simulations ignore network topology. I assume messages take 40 ns[1] to traverse the network from injection of the last byte at the source to arrival of the first at the destination. Recently, Dai and Panda [33] have shown that network contention can significantly degrade the performance of some shared-memory applications. However, because I focus on relative performance using the same base network model, I believe my quantitative results and qualitative trends can be extrapolated to more realistic networks. Additionally,

1. The SGI Spider switch, for example, offers a port-to-port latency of 40ns [41].

an abstract network model frees the evaluation from the idiosyncrasies of a particular network implementation and allows me to focus my attention purely on the NI. In Section 4.5, I study the impact of network latency on the overall performance of benchmarks.

I model hardware flow-control at the NIs using a scalable end-to-end flow control scheme called *return-to-sender* [39]. In this scheme, the sending NI allocates an empty buffer for a message and injects the message into the network. If the receiving NI has a free buffer to accept the message, it sends an acknowledgment to the sender to free up the sender's buffer. However, if the receiving NI cannot accept the incoming message due to lack of buffers, it returns the message to the sender. The sender must eject the returning message from the network into the previously allocated buffer and retry the send later. To prevent deadlock (or message loss), these returning messages must have a guaranteed path back to the sender. This can be achieved through a second network (either virtual or physical). The return-to-sender flow control strategy is scalable (unlike, for example, all-to-all buffer reservation [76]) because the number of network message buffers allocated in the NI is independent of the number of nodes in the parallel machine. In all simulations in this chapter, I assume that the number of network message buffers allocated at the sender and receiver respectively is fixed at eight. Chapter 5 examines the effect of varying this parameter on several NIs.

I ran all my benchmarks on the Tempest parallel programming interface [52]. Message-passing benchmarks use only Tempest's active messages. Shared-memory codes on Tempest also use active messages, but assume hardware support for fine-grain access control [103]. Codes with custom protocols use a combination of the two.

4.2.3 Macrobenchmarks

I use seven macrobenchmarks for evaluating the five NIs. Table 4.3 summarizes these seven macrobenchmarks—appbt, barnes, dsmc, em3d, moldyn, spsolve, and unstruc-

Benchmark	**Source**	**Input Data Set**	**Iters**	**Key Comm. Pattern**	**Msg. Size (bytes)**	**Dyn. % of msgs.**	**Dyn. % of bytes**
appbt	NASA Ames [7, 18]	24x24x24 cubes	4	Near neighbor	12 140	67% 32%	15% 85%
barnes	Stanford SPLASH-2 [134]	16K particles	4	Irregular	12 16 140	67% 4% 29%	16% 1% 82%
dsmc	U. of Maryland & Wisconsin [93]	48600 initial particles, 9720 cells	20	Fine-grain messages, producer-consumer	12 44 140	45% 25% 26%	10% 21% 68%
em3d	U. of Berkeley & Wisconsin [29, 37]	16K nodes, degree 5, 10% remote, span 2	10	Fine-grain messages	12 20	2% 98%	1% 99%
moldyn	U. of Maryland & Wisconsin [93]	2048 particles	30	Bulk reduction	8 12 140 3084	5% 65% 27% 2%	0% 7% 35% 58%
spsolve	U. of Maryland & MIT [24]	3720 elements	1	Fine-grain messages	8 12 20	6% 3% 91%	3% 2% 95%
unstructured	U. of Maryland & Wisconsin [93]	9428 nodes, 59863 edges, 5864 faces	10	Single-producer, multiple consumers	8 351 (avg.)	35% 64% (avg.)	1% 98% (avg.)

Table 4.3: Summary of macrobenchmarks. Message size includes both header and payload. Iters = number of iterations for which I simulate the macrobenchmark. Comm. = communication. Msg. = message. Dyn. = Dynamic. Avg. = average. Percentage of each benchmark may not sum to 100% due to rounding and the presence of a trivial fraction of messages of other sizes. The first six macrobenchmarks have distinct peaks at the message sizes described above. However, unstructured shows only one distinct peak at 8 bytes. Beyond that it shows a range of message sizes varying between 12-1812 bytes. Here I report the average message size for this range.

tured—and their input data sets, key communication pattern, and message size distribution. Below I describe the communication pattern of each of the seven macrobenchmarks.

Appbt is a parallel three-dimensional computational fluid dynamics application [18] from the NAS benchmark suite. It consists of a cube divided into subcubes among processors. The code is spatially parallelized in three dimensions. The main data structures are a number of 3D arrays, each of which is divided up among different processors as 3D sub-blocks. Each processor is responsible for updating the sub-block it owns. Sharing occurs between neighboring processors in 3D along the boundaries of these sub-blocks. Communication induced by sharing occurs between neighboring processors along the boundaries of the subcubes through the Wisconsin Stache protocol, which is Tempest's default invalidation-based shared memory protocol [100].

Barnes simulates the interaction of a system of bodies in three dimensions using the Barnes-Hut hierarchical N-body method [134]. The main data structure is an octree. The octree's leaves contain information about each body and internal nodes represent space cells. In each iteration the octree is rebuilt and traversed once per body to compute the forces on individual bodies. The communication pattern induced by such traversals is quite irregular.

Dsmc studies the properties of a gas by simulating the movement and collision of a large number of particles in a three-dimensional domain with discrete simulation Monte Carlo method [93]. *Dsmc* divides domains into cells in a static Cartesian grid. Each cell contains particles, which collide only with other particles in the cell. The cells are spatially divided up among processors. At the end of each iteration, particles move from one cell to another. The primary communication occurs during this movement. This chapter and Chapter 5 uses a version of dsmc that performs this communication using Tempest's active messages. Chapter 6 uses a version of dsmc that performs this communication using Stache.

Em3d models three-dimensional electromagnetic wave propagation [29]. It iterates over a bipartite graph consisting of directed edges between nodes. Each node sends two integers to its neighboring nodes through a custom update protocol [37]. Several update messages (with 12 byte payload) can be in flight, which like spsolve, can create bursty traffic patterns.

Moldyn is a molecular dynamics application, whose computational structure resembles the non-bonded force calculation in CHARMM [17]. Molecules in *moldyn* are uniformly distributed over a cuboidal region with a Maxwellian distribution of initial velocities. A molecule’s velocity and force exerted by other particles determine the molecule’s position. Force computation limits interactions to molecules within a cut-off radius. An interaction list—rebuilt every 20 iterations—records pairs of interacting molecules. The arrays that record the force exerted on molecules and molecules’ coordinates induce the maximum communication. Updates to the coordinates occur through Stache. Updates to the force array is done through a bulk reduction of the shared force array. The bulk reduction is done differently in two different versions. This chapter and Chapter 5 use a version of moldyn that perform the reduction using Tempest’s virtual channels. One execution of this reduction protocol iterates as many times as there are processors. In each of these iterations, a processor sends 3.1 kilobytes of data to the same neighboring processor through Tempest’s virtual channels. In contrast, the version used in Chapter 6 performs the bulk reduction entirely via transparent shared memory (using the Stache protocol). This reduction phase results in the migratory sharing pattern for moldyn reported in Section 6.5.

Spsolve [24] is a very fine-grained iterative sparse-matrix solver in which active messages propagate down the edges of a directed acyclic graph (DAG). All computation happen at nodes of the DAG within active message handlers. The messaging overhead is critical because each active message carries only a 12 byte payload and the total computation per message is only one double-word addition. Several active messages can be in flight, which can create bursty traffic patterns.

Unstructured is a computational fluid dynamics application that uses an unstructured mesh to model a physical structure, such as an airplane wing or body [93]. The mesh is represented by nodes, edges that connect two nodes, and faces that connect three or four nodes. The mesh is static, so its connectivity does not change. The mesh is partitioned spatially among different processors using a recursive coordinate bisection partitioner. The computation contains a series of loops that iterate over nodes, edges, and faces. Updates to nodes require a reduction phase like the one used in moldyn. Updated values of nodes are communicated along edges and faces of the mesh (in a single-producer, multiple-consumer fashion). In the version of the unstructured used in this chapter and Chapter 5, I implemented both reductions and updated using Tempest's bulk messages. However, the version of unstructured used in Chapter 6 implements both reductions and updates using Stache. Reductions result in migratory sharing and updates result in single-producer multiple-consumers sharing pattern.

4.3 Microbenchmark Results

This section examines the performance of two microbenchmarks for my five NI implementations. I simulated all four CNIs and NI_{2w} on the memory bus. However, on a coherent I/O bus[1] I simulated all but $CNI_{32}Q_m$, since $CNI_{32}Q_m$ cannot be implemented with current coherent I/O buses (Section 3.5). In Section 4.4 I will examine the performance of these NIs for the seven macrobenchmarks described in Section 4.2.3.

The microbenchmark numbers in this section include the messaging layer overhead for copying a message from the network interface to a user-level buffer, and vice versa. Thus, data begins in the sending processor's cache and ends in the receiving processor's cache, rather than simply moving from memory to memory.

1. See Section 3.5 for my definition of a coherent I/O bus.

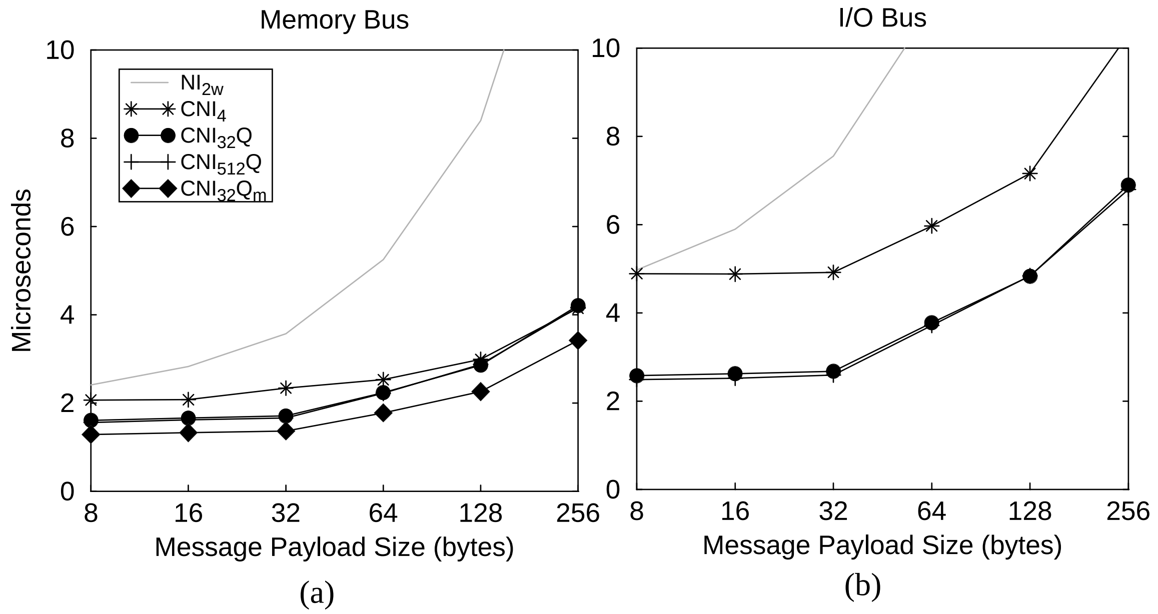
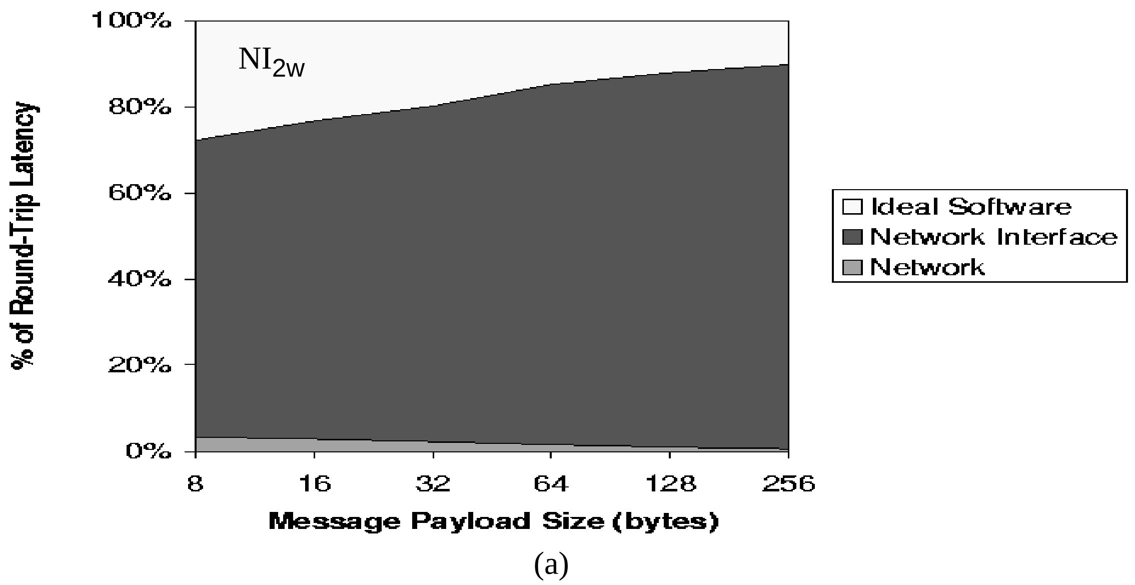
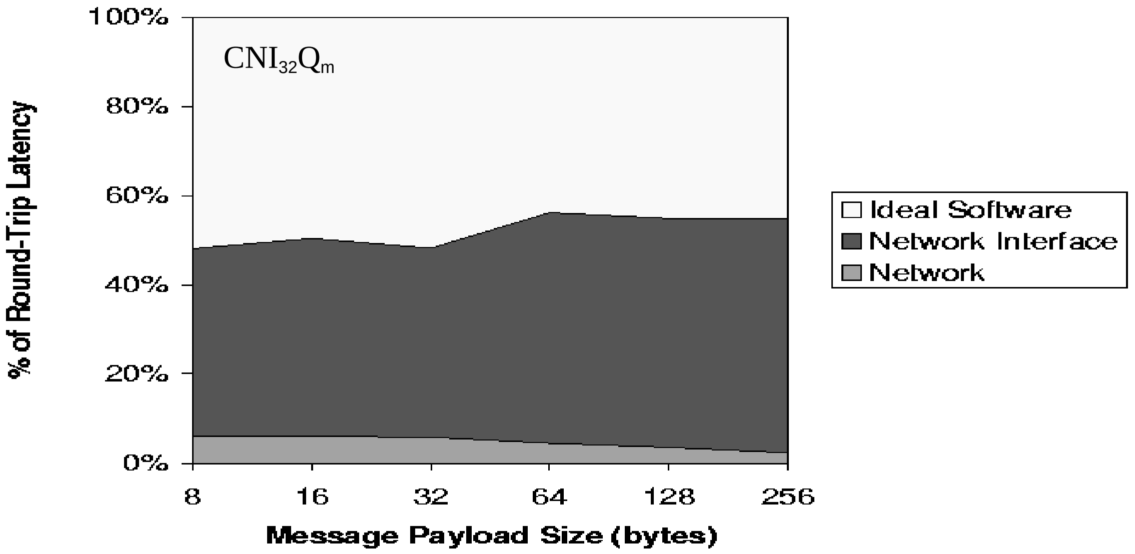


Figure 4-1. Process-to-process round-trip message latency. This figure shows the process-to-process round-trip message latency in microseconds (vertical axis) for different message sizes (horizontal axis). (a) shows the round-trip message latency for NI_{2w}, CNI_4, $CNI_{32}Q$, $CNI_{512}Q$, and $CNI_{32}Q_m$ on the memory bus. (b) shows the same (except $CNI_{32}Q_m$) on the I/O bus.

4.3.1 Round-Trip Latency

Figure 4-1 shows the round-trip latency of a message for each of NI_{2w} and the four CNIs. It shows two important results. First, CNIs reduce messaging overheads significantly. For small messages, between 8 and 256 bytes, $CNI_{32}Q_m$ is 87-342% better than NI_{2w} on the memory bus (Figure 4-1a) and $CNI_{512}Q$ is 100-377% better than NI_{2w} on the I/O bus (Figure 4-1b). Second, on the memory bus $CNI_{32}Q_m$ consistently outperforms the other three CNIs. $CNI_{32}Q_m$ outperforms CNI_4 by 20-60% because, unlike CNI_4, $CNI_{32}Q_m$ polls on a cached message valid bit and amortizes the expensive three-cycle handshake over an entire queue of messages. $CNI_{32}Q_m$ outperforms $CNI_{32}Q$ and $CNI_{512}Q$, which perform similarly, by 21-25% because $CNI_{32}Q_m$ organizes its memory as a small cache, which can be built with fast SRAMs. In contrast, I expect $CNI_{32}Q$ and $CNI_{512}Q$ to be built with slower DRAMs because of their large buffer requirements (Section 2.3).





(b)

Figure 4-2. Breakdown of round-trip latency for NI_{2w} and $CNI_{32}Q_m$. This figure breaks down the round-trip message latency of NI_{2w} and $CNI_{32}Q_m$ (attached to the memory bus) into three components—ideal software latency, network interface latency, and network. The ideal software latency is the NI-independent, minimum message protocol latency incurred with a NI_{2w} device attached directly to the processor and accessed in one cycle. The network interface latency includes both the hardware and software overhead of the specific network interface. Finally, network latency represents just the time to traverse the network. The absolute latency numbers corresponding to the percentages are available from Figure 4-2.

Figure 4-2 further shows the contribution of three components—NI-independent, minimum software protocol latency, latency to access the NI (both software and hardware), and network latency—to the total round-trip latency for NI_{2w} and $CNI_{32}Q_m$. For this experiment both NI_{2w} and $CNI_{32}Q_m$ are attached to the memory bus. I chose only NI_{2w} and $CNI_{32}Q_m$ because these are the worst- and best-performing NIs on the memory bus. I calculated the NI-independent, minimum software protocol latency by subtracting two times the network latency from the total round-trip latency for an ideal NI_{2w} device attached directly to the processor and accessed in a single cycle.

Figure 4-2 shows that $CNI_{32}Q_m$ significantly reduces the contribution of the NI access latency to the total round-trip latency. Additionally, the contribution of $CNI_{32}Q_m$'s access latency to the overall round-trip latency is almost same or lower than the NI-independent, minimum software protocol latency. This suggests that significant performance gains in latency cannot be achieved without similar reductions in the software protocol latency itself.

4.3.2 Bandwidth

Figure 4-3 graphs the bandwidth provided by the five network interfaces. This figure shows that CNIs improve the bandwidth over NI_{2w} significantly, *even for very small messages*. On the coherent memory bus, $CNI_{32}Q_m$ is 109-202% better than NI_{2w} for 8-4096 byte messages (Figure 4-3a). For the same message sizes, $CNI_{512}Q$ is 113-402% better than NI_{2w} on the coherent I/O bus (Figure 4-3b).

All four CNIs offer significantly greater bandwidth than NI_{2w}. Among the CNIs, CNI_4 performs worst of the four CNIs because of its high overhead for polling uncached registers and the three-cycle handshake in the critical path of message reception. $CNI_{32}Q$ and $CNI_{512}Q$ perform the best due to their low poll overhead and ability to cache multiple messages (a network message fits in four cache blocks).

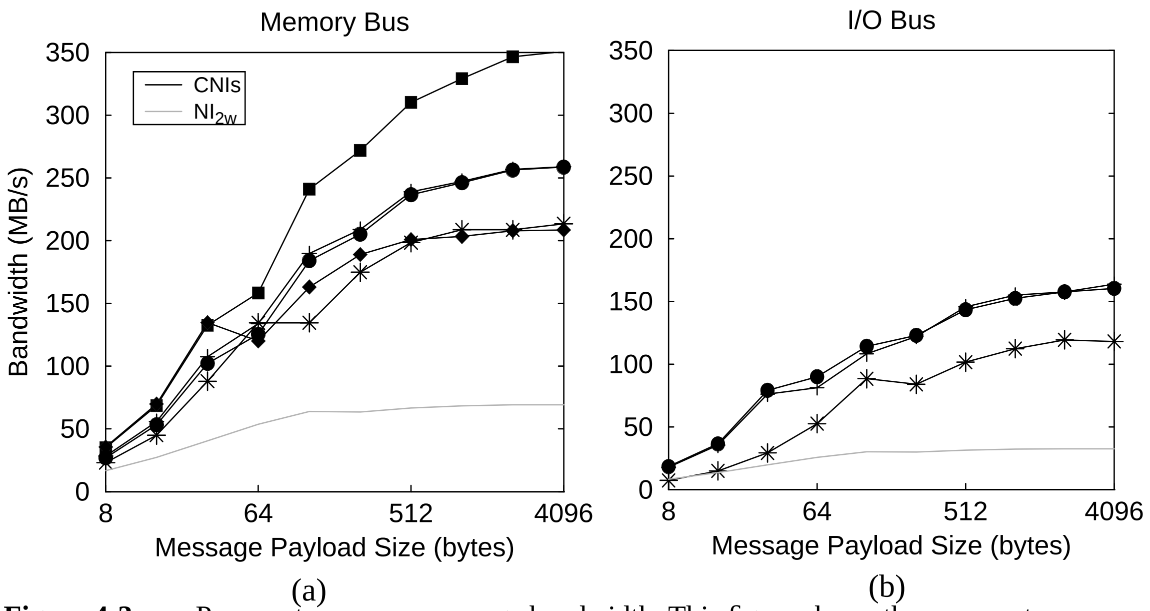


Figure 4-3. Process-to-process message bandwidth. This figure shows the process-to-process message bandwidth in megabytes per second (vertical axis) for different message sizes (horizontal axis). (a) shows the process-to-process message bandwidth for NI_{2w}, CNI_4, $CNI_{32}Q$, $CNI_{512}Q$, $CNI_{32}Q_m$, and $CNI_{32}Q_m$ with send throttling. (b) shows the same (except the $CNI_{32}Q_m$ graphs). CNI Legend: ■ = $CNI_{32}Q_m$+send throttle, ◆ = $CNI_{32}Q_m$, + = $CNI_{512}Q$, ● = $CNI_{32}Q$, ✻ = CNI_4.

CNI_4 shows two different knees on the coherent memory and I/O buses respectively. The knee on the I/O bus appears when CNI_4 saturates the I/O bus. However, the cause of the knee on the memory bus is slightly more subtle.

CNI_4's knee on the memory bus appears when the message size exceeds one cache block. This is because before the CNI_4 device can write a new message, it must invalidate the corresponding CDR in the processor's cache. Each CDR requires a separate invalidation signal on current memory buses. However, before the CNI_4 device completes invalidating all the CDRs, the processor's uncached poll for the receive status register completes and reports no pending message in the CNI_4 device. This forces the processor to exit the optimized inner loop that extracts messages from the NI and processes them. When the CNI_4 device completes writing new messages to the CDRs, the processor re-enters the inner message processing loop, which incurs additional overhead, such as checking and setting interrupt masks. This effect does not occur for messages less than a cache block, because the CNI_4 device manages to grab the memory bus and invalidate the single CDR block before the processor's uncached poll appears on the memory bus. Interestingly, this knee does not appear on the I/O bus, because the processor's uncached load for the status register is repeatedly nack-ed by the I/O bridge until all the coherent invalidates issued by CNI_4 have been satisfied. By the time the processor's pending uncached load acquires the I/O bus and completes the transaction, the CNI_4 device has already successfully written a new message to the corresponding CDRs. Consequently, the processor does not exit the inner message processing loop in this case.

$CNI_{32}Q_m$ achieves slightly lower bandwidth than $CNI_{512}Q$. This is because the message send rate is significantly greater than the message reception rate, causing the receiving $CNI_{32}Q_m$'s cache to overflow. The resulting writebacks to main memory induce heavy bus contention, which decreases the maximum communication bandwidth. Unfortunately, because the problem is bandwidth not latency, a writeback buffer will not help with this microbenchmark as it would for the round-trip microbenchmark. However, throttling the sender appropriately, can significantly increasing the bandwidth. Figure 4-3a shows, throttling the $CNI_{32}Q_m$ sender can increase the bandwidth for 4096-byte messages from 209 MB/s to around 351 MB/s.

4.4 Macrobenchmark Results

This section evaluates the performance of the NIs described in Section 4.1 with seven macrobenchmarks. The next chapter (Chapter 5) delves deeper into two key parameters—data transfer and buffering—that impact the design of high-performance NIs and compares the performance of $CNI_{32}Q_m$ and $CNI_{512}Q$ with five other network interfaces (including NI_{2w}).

Section 4.4.1 compares the performance of the five NIs on coherent memory and I/O buses. Section 4.4.2 shows the speedup achieved by the seven macrobenchmarks with $CNI_{32}Q_m$, which performs the best on the coherent memory bus, and $CNI_{512}Q$, which performs the best on the coherent I/O bus.

Macrobenchmark	appbt	barnes	dsmc	em3d	moldyn	unstuctured	spsolve
I/O Bus Slowdown	0.62	0.71	0.79	0.60	0.67	0.58	0.74

Table 4.4: NI_{2w}'s slowdown on the I/O bus. This table shows NI_{2w}'s execution time on the memory bus divided by NI_{2w}'s execution time on the I/O bus for the seven macrobenchmarks. These normalizing factors should be used to compare Figures 4-4 and 4-5, which compare the performance of these macrobenchmarks on the memory and I/O buses respectively.

4.4.1 Performance Comparison of five NIs on Memory and I/O Buses

Figures 4-4 and 4-5 show the performance gains from CNIs on coherent memory and I/O buses respectively for the seven macrobenchmarks. Both these figures are normalized to the execution time of NI_{2w} on the respective buses. Table 4.4 provides the normalizing factors that must be used for each macrobenchmark to compare these figures. On the memory bus, I examine all five NIs described in Section 4.1. On the coherent I/O bus, I examine all, but $CNI_{32}Q_m$, because $CNI_{32}Q_m$ cannot be implemented with current I/O buses.

CNI_4, $CNI_{32}Q$, $CNI_{512}Q$, and $CNI_{32}Q_m$ offer a progression of incremental benefits over NI_{2w}. Unlike NI_{2w}, which can only be accessed through uncached loads and stores, CNI_4 effectively exploits the memory bus's high-bandwidth block transfer mechanism by transferring messages in full cache block units. $CNI_{32}Q$ and $CNI_{512}Q$ further improve performance by polling for incoming messages on a cachable memory location, amortizing the three-cycle handshake over an entire queue of messages, and providing a larger capacity for messages that helps prevent bursty traffic from backing up the network. $CNI_{32}Q_m$ further simplifies software flow control in the messaging layer by allowing messages to smoothly overflow to main memory when the device cache fills. This avoids processor intervention for message buffering, which, otherwise, can significantly degrade performance.

Block Transfer. The increase in bandwidth obtained by transferring messages in whole cache block units has a major impact on performance. Moldyn and unstructured primarily

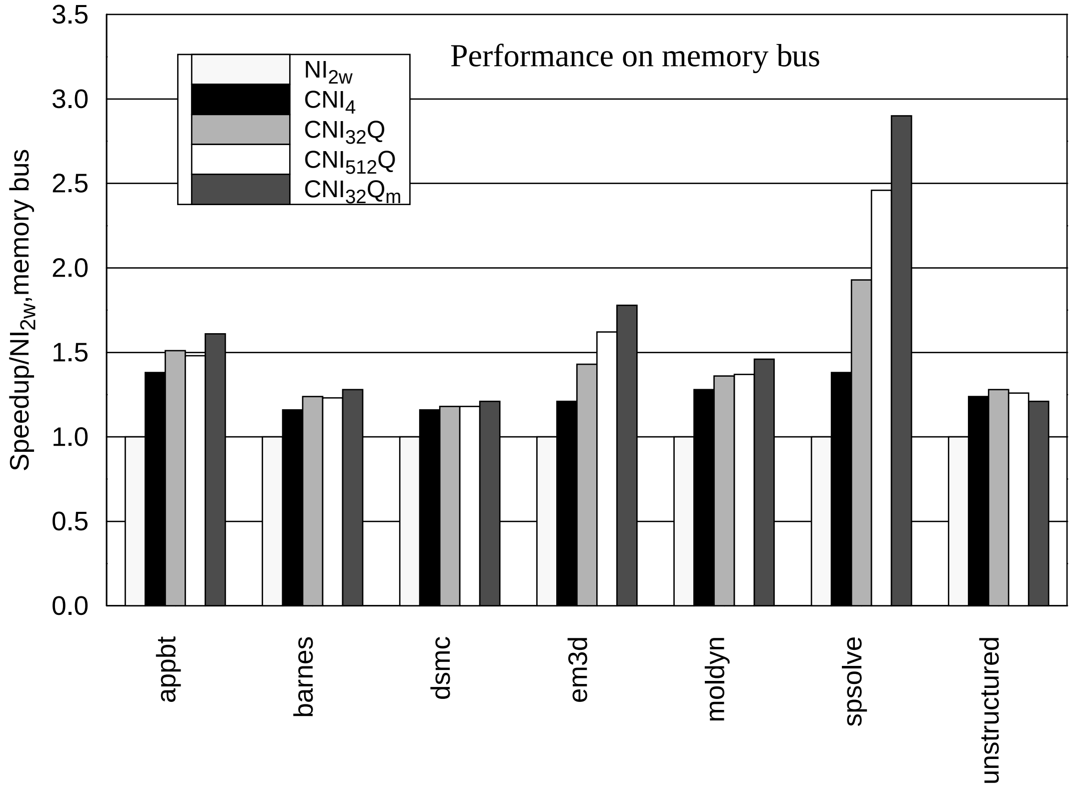


Figure 4-4. CNIs' performance on the memory bus. This figure compares the performance of NI_{2w}, CNI_4, $CNI_{32}Q$, $CNI_{512}Q$, and $CNI_{32}Q_m$ on the memory bus for seven macrobenchmarks. The vertical axis shows the speedup of each of the NIs over NI_{2w}.

do bulk transfers and appbt and barnes communicate with moderately large (128-byte) shared-memory blocks. Moldyn's reduction protocol transfers 3.1 KB of data between neighboring processors, while two processors in unstructured, on average, exchange 351 bytes between them in the main communication phases (see Table 4.3 for message size distributions). Consequently, on average, the portion of a CDR block that does not carry useful information is moderately low—30% for moldyn and 14% for unstructured (Table 4.5). For appbt and barnes, roughly half a cache block space is wasted on average.[1] However, the time wasted by this under-utilization is low because today's memory buses are often as wide as half a cache block (e.g., SUN Ultragigaplane [112]), which can be transferred over the memory bus in only two memory bus cycles.

1. This problem may not arise with variable-sized message entries in which a new message starts immediately after the end of the previous message. However, this may introduce new problems, such as false sharing.

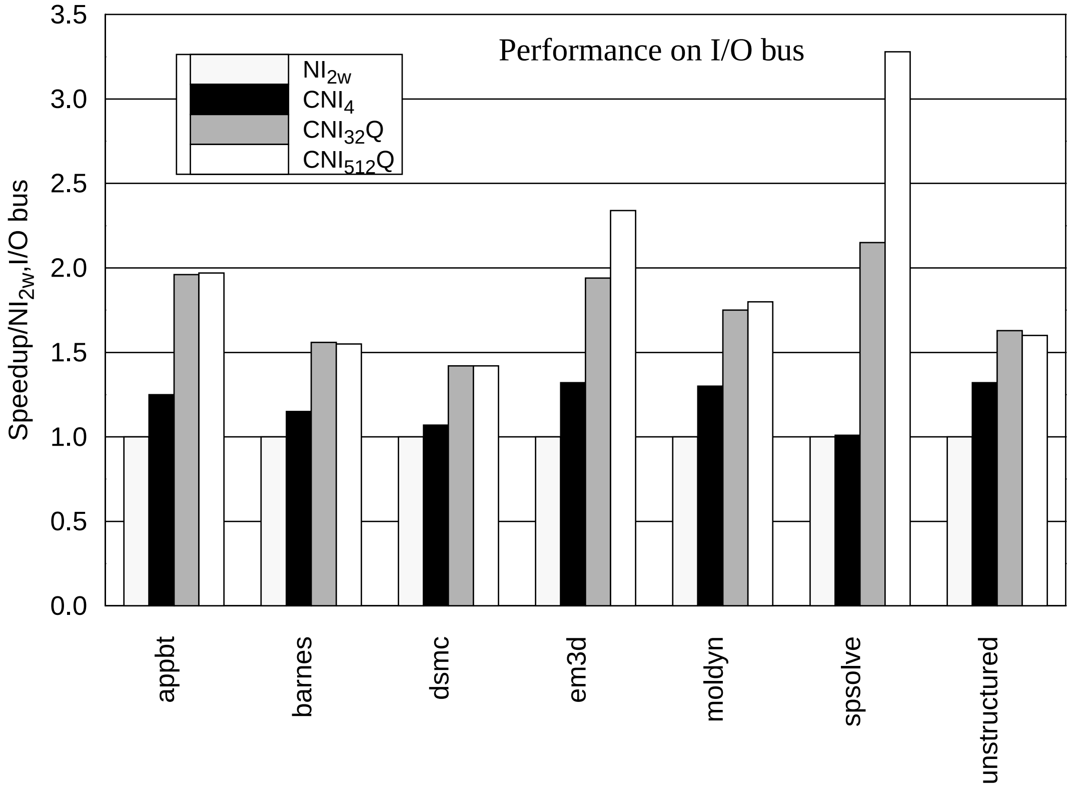


Figure 4-5. CNIs' performance on the I/O bus. This figure compares the performance of NI_{2w}, CNI_4, $CNI_{32}Q$, and $CNI_{512}Q$ on the I/O bus for seven macrobenchmarks. The vertical axis shows the speedup of each of the NIs over NI_{2w}.

With respect to actual execution time, CNI_4 improves moldyn, unstructured, appbt, and barnes' performance by 28%, 24%, 38%, and 16% respectively on the memory bus. For the I/O bus, the improvements for these macrobenchmarks are 30%, 32%, 25%, and 15% respectively. Even for em3d and spsolve that send small messages (12-byte payload) and dsmc that communicates with relatively small (44-byte) messages and moderately large (128-byte) shared-memory blocks, CNI_4's performance improvement over NI_{2w} is significant—that is, between 7-38%—except for spsolve on the I/O bus, which shows only a 1% improvement of NI_{2w}.

CNI_4's performance improvement for spsolve on the coherent I/O bus is not as high as on the memory bus because of contention at the I/O bridge. The NI_{2w} device never tries to acquire the memory or I/O bus because it is always a bus slave. However, CNI_4 competes

Macrobenchmarks	**Percentage of cache block wasted**
appbt	45.70
barnes	47.50
dsmc	41.20
em3d	63.61
moldyn	29.54
spsolve	63.85
unstructured	13.57

Table 4.5: Percentage of a cache block space wasted for CNI_4. This table shows the percentage of a CDR block that does not carry useful information when transferring data between CNI_4 and a processor's cache. The CQ-based CNIs—$CNI_{32}Q$, $CNI_{512}Q$, and $CNI_{32}Q_m$—show similar utilization for the CQ blocks.

with the processor cache to acquire the memory and I/O buses. Simultaneous bus acquisition requests at the I/O bridge from the processor cache and CNI_4 cache creates contention. This effect appears to be severe in spsolve. For example, the memory bus occupancy (Table 4.7) for a system with CNI_4 on the I/O bus compared to a system with NI_{2w} on the I/O bus decreases between 10-44% for all macrobenchmarks, except spsolve for which the occupancy increases by 3%.

Overall, for the seven macrobenchmarks, CNI_4 improves the performance over NI_{2w} between 16-38% on the coherent memory bus and 1-32% on the coherent I/O bus. Intra-message prefetching (Section 3.2.2) accounts for between 0-8% of this improvement on the memory bus. In one case—em3d on the memory bus—intra-message prefetching actually deteriorates performance by 2%. Intra-message prefetching does not help on the I/O bus.

On the memory bus CNI_4's performance improvement accounts for 58% (averaged across all macrobenchmarks) of the total gain achieved by $CNI_{32}Q_m$, which performs the best on the memory bus. On the I/O bus CNI_4's performance improvement accounts for

Macrobenchmark	**NI_{2w}**	**CNI_4**	**$CNI_{32}Q$**	**$CNI_{512}Q$**	**$CNI_{32}Q_m$**
em3d	29%	14%	21%	0%	0%
spsolve	55%	53%	32%	0%	0%

Table 4.6: Percentage of messages buffered explicitly for memory bus NIs.

26% (averaged across all macrobenchmarks) of the total gain achieved $CNI_{512}Q$, which performs the best on the I/O bus. The amount of buffering plays a more important role on the I/O bus because of its longer latencies, and hence block transfers (as in CNI_4) account for a lesser fraction of the total improvement on the I/O bus.

Impact of CQs. My implementations of the CQ-based CNIs improve performance over CNI_4 in three ways. First, they poll for incoming messages on a cached message valid bit. Second, they amortize the expensive three-cycle handshake over an entire queue of messages. Third, they provide extra buffering that helps smooth out bursts in message traffic. Below I examine the effect of each of these optimizations.

The cached message valid bit offered by CQs improves performance in two ways. First, in the absence of any message a processor's polls for incoming messages are satisfied directly from the processor's cache. Second, when a message does arrive, the processor's cache miss for the message valid bit brings in the first cache block of the message along with the message status. Figure 4-6, which breaks down the different NI-related memory bus transactions for the five NIs, shows that the CQ-based CNIs reduces the processor's poll transactions by a factor of 14 or more. This does not directly translate into performance for my evaluation because I have only one processor per node of the parallel machine. However, for parallel machines built with SMP nodes, this reduction can have a significant impact on performance.

Figure 4-6 also shows that the CQ-based CNIs successfully amortize the overhead of the three-cycle handshake. This is because $CNI_{32}Q$ and $CNI_{512}Q$'s non-poll transactions between the CPU and NI are consistently less than that of CNI_4. For example, $CNI_{32}Q$

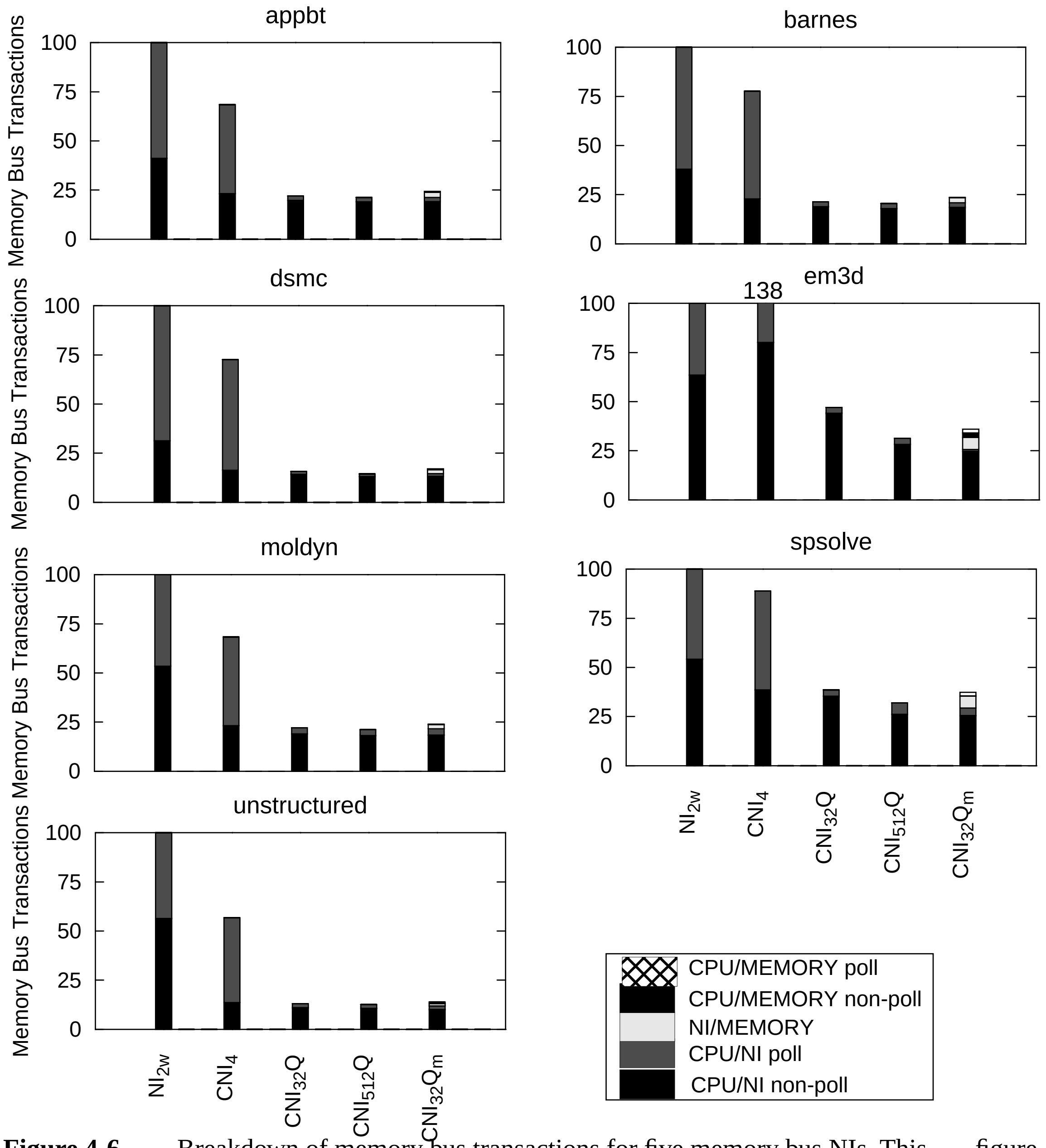


Figure 4-6. Breakdown of memory bus transactions for five memory bus NIs. This figure shows the breakdown of memory bus transactions for five memory bus NIs totalled across all 16 nodes. The notation X/Y denotes that the transaction happened between modules X and Y on the memory bus. For NI_{2w} and CNI_4, poll denotes transactions corresponding to a processor's polls to the send and receive status registers. For CQ-based CNIs, poll denotes the processor's cache misses for cache blocks that return without valid data and the processor's polls for the send status register, which is uncached in these CNI implementations. Non-poll denotes all NI-related transactions other than polls. CNI_4 with em3d shows a slight anomaly because node 15 became a hot spot due to accidental differences in global memory allocation. However, this appears to have only significantly affected the bus transaction count, and not the total execution time of CNI_4.

reduces the number of non-poll memory bus transactions between 8-45% over CNI_4 on the memory bus.

The extra buffering provided by $CNI_{32}Q$ and $CNI_{512}Q$ helps smooth out bursty traffic patterns and avoids clogging up the network. In my simulated system, if a processor blocks on a send due to the absence of adequate buffering in the message path, then the processor explicitly removes messages from the NI and stores them in buffers allocated in the user's virtual memory. This is necessary to avoid deadlock in the software message library [68]. The reduction in number of messages buffered explicitly in user's virtual memory is an indirect and approximate measure of the potential improvement offered by CQ's extra buffering capability.

Table 4.6 lists the percentage of messages buffered by the five NIs on the memory bus for em3d and spsolve. For the other macrobenchmarks, the extra buffering has a small impact on performance (less than 10%). As Table 4.6 shows, the extra buffering provided by $CNI_{32}Q$ and $CNI_{512}Q$ does indeed help reduce the number of messages buffered explicitly for em3d and spsolve. Table 4.6, however, shows the same anomalous situation for CNI_4 with em3d, as was shown in Figure 4-6. Em3d sends almost twice as many messages for CNI_4 compared to NI_{2w}, even though the number of messages buffered explicitly is almost same. This effect reduces the overall percentage of messages buffered from 29% in NI_{2w} to 14% in CNI_4.

Overall, for em3d on the memory bus, $CNI_{32}Q$ and $CNI_{512}Q$ improves performance over CNI_4 by 18% and 40% respectively. For spsolve, the corresponding numbers are 34% and 78% respectively. The long latencies due to the I/O bus, however, reduce the rate at which messages can be removed from an I/O bus NI. Consequently, extra buffering has a greater impact on performance. Across the seven macrobenchmarks, $CNI_{32}Q$ and $CNI_{512}Q$ improve performance over CNI_4 between 13-57% and 21-77% respectively.

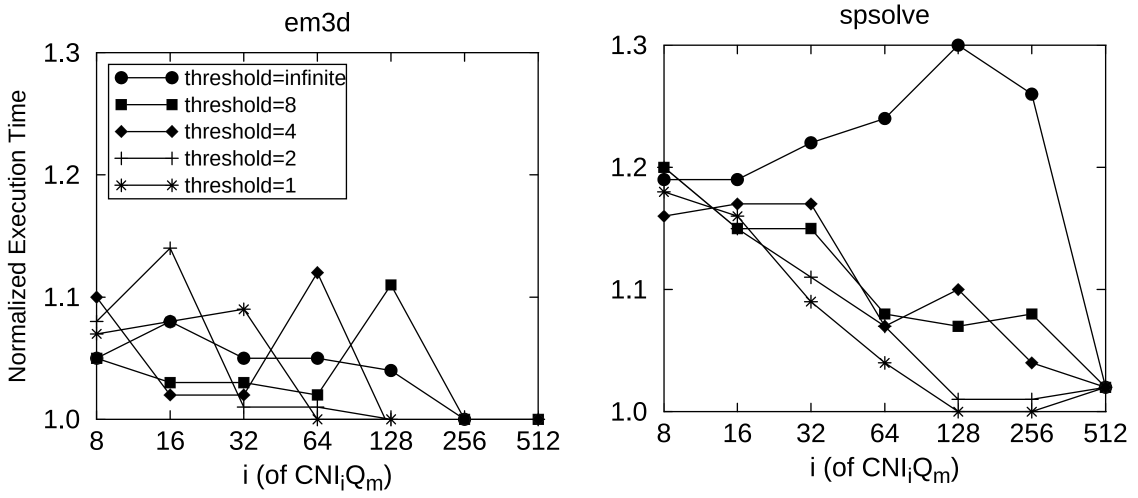


Figure 4-7. CNI$_i$Q$_m$ cache size variation. This figure shows the effect of varying CNI$_i$Q$_m$'s cache size, i.e. "i", and the dead message elimination threshold described in Section 3.3.2. Each graph is normalized to the execution time of CNI$_{128}$Q$_m$ with threshold = 1. An infinite threshold indicates no dead message elimination optimization.

Overflow to Memory. CNI$_{32}$Q$_m$ allows messages to smoothly overflow to memory when the device cache fills up. Figure 4-4 shows that CNI$_{32}$Q$_m$ performs similarly to CNI$_{512}$Q, except for spsolve. For spsolve, CNI$_{32}$Q$_m$ outperforms CNI$_{512}$Q by 18% because of its faster memory (Table 4.2). However, if the CNI$_{512}$Q memory can offer the same latency as in CNI$_{32}$Q$_m$, then the performance of both NIs are almost similar for all the seven macrobenchmarks (not shown). Thus, CNI$_{32}$Q$_m$ performs similarly or better than CNI$_{512}$Q *even with significantly less memory (i.e., cache) on the device.*

Figure 4-7 studies the effect of varying the cache size, i.e. "i", for CNI$_i$Q$_m$. The variation in cache size has a major impact only on the performance of em3d and spsolve—the two macrobenchmarks for which buffering plays an important role. For other macrobenchmarks the effect of cache size variation is less prominent and the variation in performance is within at most a 6% range of CNI$_{32}$Q$_m$.

As Figure 4-7 shows, the dead message elimination threshold (see Section 3.3.2) interacts with the variation in CNI$_i$Q$_m$ cache size. With an infinite threshold—that is, the receive CQ's head is always read lazily—CNI$_i$Q$_m$ is most of the time unsuccessful in determining if messages are already dead, and consequently it ends up flushing dead messages to main memory. This effect is particularly severe in spsolve, where performance is affected heavily. However, when the cache size equals the memory allocated for the queue—512 memory blocks in this case—the entire queue fits in the cache, and CNI_iQ_m does not have to flush any message to memory.[1]

The flush threshold for which CNI_iQ_m offers the best performance varies with the cache size. For cache sizes greater than 64 blocks, a flush threshold of one offers the best performance for both em3d and spsolve. For cache sizes of 32 cache blocks or less, the best flush threshold varies, but in a very narrow range. Hence, for all my experiments (except Figure 4-7) with CNI_iQ_m, I have assumed a flush threshold of one.

Section 3.3.2, which describes the dead message elimination optimization, also proposes a second optimization called cache bypass for the case when messages are not dead and the receive cache overflows. For $CNI_{32}Q_m$, this optimization improves em3d and spsolve's performance by 4% and 8%, respectively. The fraction of messages that bypass the $CNI_{32}Q_m$ cache are 66% and 33% for em3d and spsolve respectively. All graphs in this and the next chapters assume cache bypass optimization for CNI_iQ_m.

Overall, $CNI_{32}Q_m$ shows the best performance improvement over NI_{2w} (between 21-190%) on the coherent memory bus, and $CNI_{512}Q$ shows the best performance improvement over NI_{2w} (between 42-228%) on the coherent I/O bus.

Finally, CNIs significantly reduce the memory bus occupancy. Table 4.7 shows the memory bus occupancy for NI_{2w} and $CNI_{32}Q_m$, both attached to the memory bus. $CNI_{32}Q_m$ reduces the occupancy of a non-split-transaction memory bus, as I have assumed for this evaluation, by 58-78%. Because this may overestimate the bus occupancy reduction for a split-transaction bus, Table 4.7 characterizes the bus occupancy reduction for bus arbitration and data transfer, which cannot be avoided even in a split-transaction bus. Table 4.7 shows the $CNI_{32}Q_m$ reduces the arbitration cycles between 38-67% and

1. CNI_iQ_m still flushes the sense bits, even though it is unnecessary.

Macrobenchmark	NI_{2w}			$CNI_{32}Q_m$		
	Total	Arb	Data	Total	Arb	Data
appbt	15.20	1.00	0.99	4.27	0.56	0.52
barnes	17.56	1.00	1.10	7.29	0.60	0.67
dsmc	18.81	1.00	1.06	6.17	0.44	0.48
em3d	20.21	1.00	1.06	7.70	0.62	0.62
moldyn	16.12	1.00	1.00	3.53	0.50	0.46
spsolve	18.88	1.00	1.05	4.50	0.61	0.52
unstructured	19.63	1.00	1.08	6.60	0.33	0.41

Table 4.7: Breakdown of memory bus cycles for NI_{2w} and $CNI_{32}Q_m$. This table shows the breakdown of the memory bus cycles for the non-split-transaction memory bus I used for all my simulations. Total denotes the total bus occupancy, arb denotes the cycles required for bus arbitration, and data denotes the cycles required for data transfer. The table is normalized to NI_{2w}'s arbitration cycles.

data transfer cycles between 39-62%. Thus, $CNI_{32}Q_m$ frees up at least a third of NI_{2w}'s memory bus bandwidth for use by other processors.

4.4.2 $CNI_{32}Q_m$ and $CNI_{512}Q$ Speedup

Figure 4-8 plots the speedup curves of the seven macrobenchmarks with $CNI_{32}Q_m$ on the coherent memory bus and $CNI_{512}Q$ on the coherent I/O bus. These NIs perform the best on the respective buses.

$CNI_{32}Q_m$ on the coherent memory bus and $CNI_{512}Q$ on the coherent I/O bus show moderate to good speedup for all macrobenchmarks, except spsolve. For 16 nodes, $CNI_{32}Q_m$'s speedup on the memory bus ranges between 4.5 and 13.9, while $CNI_{512}Q$'s speedup on the I/O bus ranges between 3.87 and 10.91.

With 16 nodes, spsolve's speedup is only 1.2 for $CNI_{32}Q_m$ on the coherent memory bus. For 16 nodes, spsolve shows no speedup for $CNI_{512}Q$ on the coherent I/O bus. However,

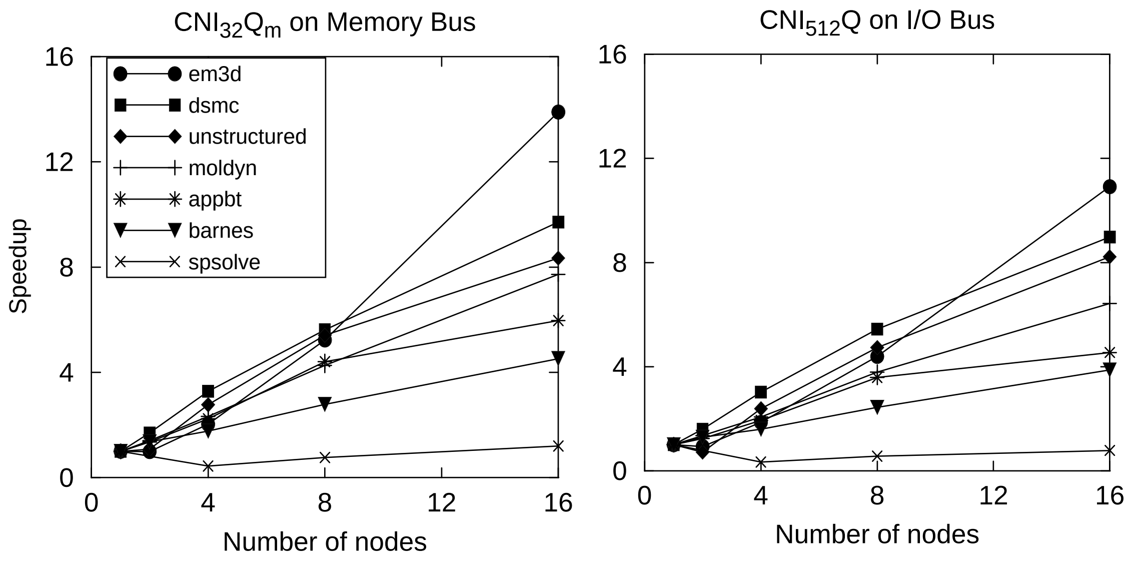


Figure 4-8. Macrobenchmark speedup. This figure shows the speedup of the seven macrobenchmarks with $CNI_{32}Q_m$ on the memory bus and $CNI_{512}Q$ on the I/O bus. These NIs perform the best on the respective buses.

for $CNI_{512}Q$ on the I/O bus, spsolve achieves a speedup of 1.29 for 32 nodes (not shown in graph). Although parallel spsolve does not show spectacular speedup, it can still be useful. This is because of two reasons. First, spsolve is primarily used as an embedded kernel in parallel applications [24]. Consequently, according to Amdahl's Law, any speedup obtained from spsolve is beneficial. Second, data required by different processors in spsolve may already be distributed across different nodes of the parallel machine. This may make it difficult and/or expensive to run the sequential version of spsolve, instead of the parallel version.

A speedup of four or more—as shown by most of my macrobenchmarks—is actually cost-effective on today's cluster of workstations. Wood and Hill argue [137] that parallel computing is cost-effective, when speedups of applications exceed the costup of the parallel machine. Wood and Hill define costup as the ratio of the cost of the parallel machine vs. cost of a uniprocessor machine. For an older generation of SGI machines, Wood and Hill showed that for parallel applications with large memory requirements, parallel computing can be cost-effective with speedups much less than linear.

An examination of the cost of Sun Enterprise E3000 servers, Ultra1 workstations, and Myricom Myrinet network reveals that parallel applications need achieve only a speedup of two to make a 16-node parallel machine cost-effective. A uniprocessor Sun E3000 enterprise server—with 4 GB of memory and a 167 MHz UltraSPARC processor—costs $232,500 [82]. In comparison, a 16-node parallel machine—built with 16 Ultra1 workstations, each equipped with 512 MB of memory and a 167 MHz UltraSPARC processor, and connected with a Myricom Myrinet network—costs $452,960. This is because each Ultra1 workstation costs $26,495 [81] and a 16-node Myricom Myrinet network costs $29,040 [26]. The total memory on the 16-node parallel machine is 8 GB—twice that of the uniprocessor E3000 machine—to allow for any extra memory that parallel applications may require. I cannot use a uniprocessor Ultra1 workstation as the uniprocessor machine, because an Ultra1 can only support upto 2 GB of memory. Thus, the costup of the parallel machine is slightly less than two ($452,960 / $232,500), and probably even less because the cost of each Ultra1 workstations above includes the cost of a color monitor. Consequently, for such a system, parallel applications need achieve a speedup of only two to make parallel computing cost-effective. In contrast, most of my macrobenchmarks achieve a speedup of four or more on a 16-node parallel machine.

4.5 Impact of Network Latency

Finally, Figure 4-9 shows the impact of network latency on CNI techniques. On the memory bus, $CNI_{32}Q_m$ consistently outperforms NI_{2w} below a network latency of 10 microseconds. Beyond 10 microseconds the relative importance of CNI techniques decreases because the network becomes the dominant bottleneck.

4.6 Conclusions

This chapter evaluates the performance of four alternate CNI designs—CNI_4, $CNI_{32}Q$, $CNI_{512}Q$, and $CNI_{32}Q_m$—with a CM-5-like NI. Microbenchmark results showed that CNIs significantly improved the round-trip latency and bandwidth of small and moderately large messages. For small message sizes, between 8 and 256 bytes, CNIs improved

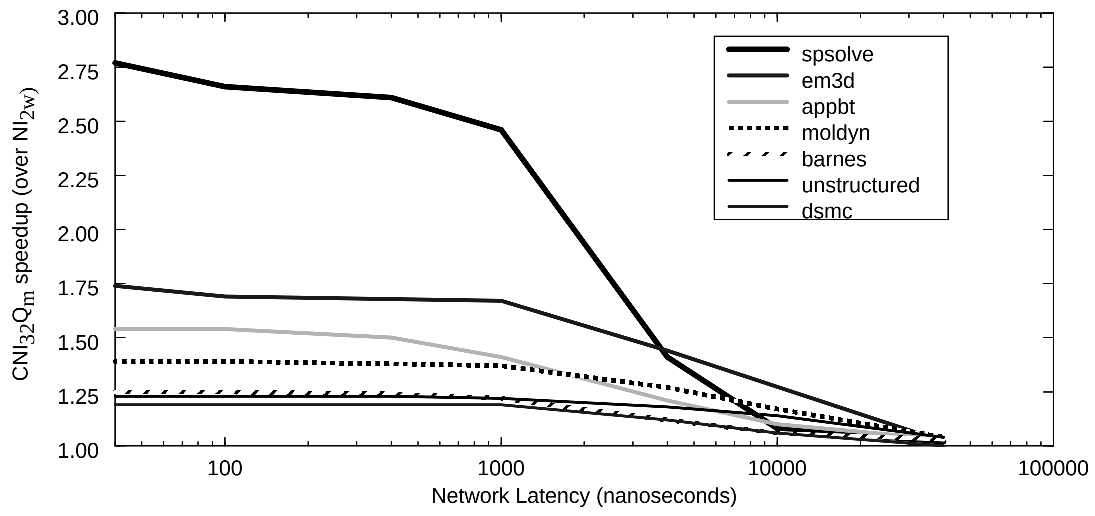


Figure 4-9. Impact of Network Latency.

the round-trip latency by 87-342% compared to NI_{2w} on a coherent memory bus and 100-377% on a coherent I/O. For moderately large messages, between 8 and 4096 bytes, CNIs improved bandwidth by 109-202% over NI_{2w} on a coherent memory bus and 113-402% on an I/O bus.

Macrobenchmark results showed that $CNI_{32}Q_m$ performed the best on the coherent memory bus and $CNI_{512}Q$ on the coherent I/O bus. $CNI_{32}Q_m$ was 21-190% better than NI_{2w} on the memory bus, while $CNI_{512}Q$ was better than NI_{2w} by 42-228% on the I/O bus. This performance boost from $CNI_{32}Q_m$ and $CNI_{512}Q$ come from their ability to rapidly transfer data in cache block units over the memory bus, provide low-overhead cachable queues, and plentiful buffering either in main memory or in the NI itself. The next chapter examines the data transfer and buffering parameters more carefully, and compares $CNI_{32}Q_m$ and $CNI_{512}Q$ with five alternative NIs, all attached to the memory bus.

Chapter 5

Impact of Data Transfer and Buffering Alternatives

This thesis examines how processor accesses to a network interface (NI) can be significantly accelerated by treating accesses to the network interface as memory accesses. Chapter 2 argued that such treatment opens up at eight opportunities for performance improvement. Chapter 3 proposed and Chapter 4 evaluated a novel class of NIs called Coherent Network Interfaces that exploit all the eight opportunities. This chapter systematically examines and evaluates two of the eight parameters—data transfer and buffering—in greater detail.

The data transfer parameters capture how messages are transferred between a processor and an NI. The buffering parameters capture where and how an NI buffers incoming network messages. Figure 5-1 shows the impact of data transfer and buffering parameters on the performance of seven parallel scientific applications studied in this chapter. This figure shows that data transfer and buffering can respectively account for up to 42% and 58% of the total execution time of these parallel programs. In other words, proper choices of the data transfer and buffering parameters can have a dramatic impact on performance.

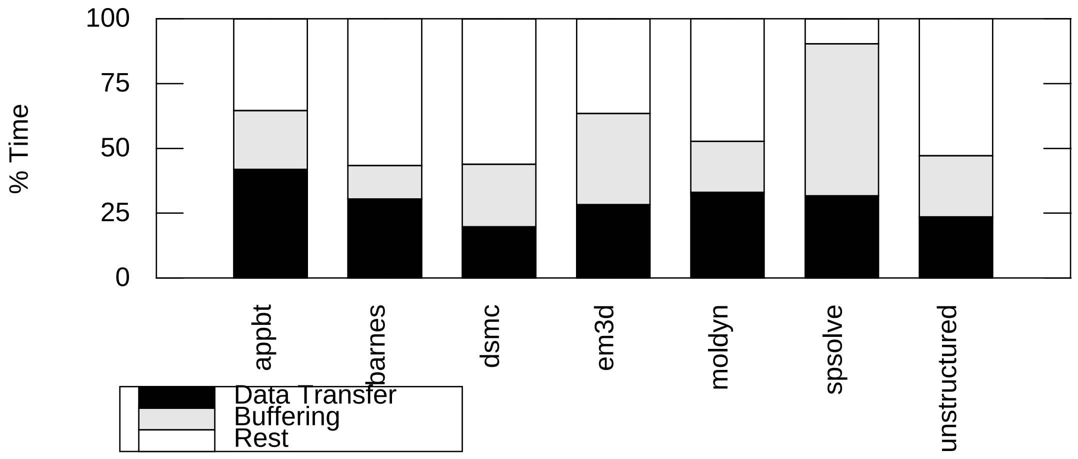


Figure 5-1. Impact of data transfer and buffering. This figure demonstrates the impact of data transfer and buffering on the performance of a memory bus NI for seven parallel scientific applications. For these measurements, I use a CM-5-like network interface and number of flow control buffers equal to 1. See Section 5.3 and Section 5.4 for a description of my CM-5-like NI, my flow control scheme, and the applications.

The data transfer and buffering parameters expose an enormous NI design space. This is because these parameters can be implemented in several different ways. In this chapter I evaluate seven memory bus NIs that I believe capture the essential components of this design space. These seven memory bus NIs abstract the key data transfer and buffering parameters of the NIs for the TMC CM-5 [68], Fujitsu AP3000 [111], Princeton User-Level DMA [11], Digital Memory Channel [44], MIT StarT-JR [53], and two CNIs—$NI_{512}Q$ and $CNI_{32}Q_m$—described in Chapter 3.

I evaluate these NIs using the same two microbenchmarks and seven macrobenchmarks described in Section 4.2. My results indicate that a high-performance NI design must:

- effectively use the block transfer mechanism of current memory buses,
- minimize processor involvement for data transfer,
- directly transfer messages between an NI and a processor, at least in the common case,
- provide plentiful buffering, possibly in main memory, and
- minimize processor involvement to buffer incoming network messages.

The relative importance of these parameters depends on both the specific NI design and the characteristics of the application.

These observations are, however, applicable primarily to user-level NIs [88] targeted for fine-grain communication. NIs that require operating system intervention for message send and receipt or must transfer multi-megabytes of data directly from a graphics device or a disk (e.g. in a video server) may require optimizations that are different from those discussed in this chapter.

I have two main contributions in this chapter. First, I identify and examine the key data transfer and buffering parameters that underlie high-performance, user-level NI designs for fine-grain communication. Second, I undertake the first systematic simulation study that compares seven NIs representative of the design space exposed by these parameters. As a corollary of this study, I find that, contrary to conventional wisdom, mapping an NI to the processor registers may not be the ideal choice. This is because processor register memory is a precious resource, which may not provide adequate buffering for some applications.

The rest of the chapter is organized as follows. Section 5.1 and Section 5.2 discuss the different data transfer and buffering parameters. Section 5.3 describes the seven memory bus NIs I studied in this chapter. Section 5.4 describes my evaluation methodology. Section 5.4 discusses my results. Section 5.5 describes related work. Finally, Section 5.6 presents my conclusions.

5.1 Data Transfer Parameters

An NI is a device that sends and receives messages to and from an external network on behalf of the processor. Consequently, the most important data sent and received by an NI are network messages. For high performance, NIs must transfer these messages rapidly between the internal memory structures (e.g. processor registers, main memory) of a node and the NI. For memory bus NIs (Figure 2-1), such data transfer occurs over the memory bus.

I have identified and will discuss three key parameters that influence the speed of such data transfer:

- size of transfer
- degree of processor involvement for transfer, and
- source and destination of transfer.

5.1.1 Size of Transfer

Today most high-performance memory buses allow at least two data transfer sizes: small chunks (between one to eight bytes) and medium-sized blocks (between 16-64 bytes). The latter is more efficient than the former mechanism because block transfers can effectively use the bandwidth available from wide memory buses and amortize control overheads, such as bus arbitration, grant, and turnaround time.

Several recent studies show that applications can effectively use such block transfers over the memory bus. Clearly, if the typical message size in fine-grain communication were a few bytes, block transfers over the memory bus would be useless. However, Cypher, et al. [32] found that in seven parallel scientific applications 30% of the messages were between 16 bytes and a kilobyte. Kay and Pasquale [59] found that the median message sizes for TCP and UDP (mostly generated by the Network File System) traffic in a departmental network were 32 and 128 bytes respectively. They also found that 99% of TCP and 86% of the UDP traffic was less than 200 bytes. Keeton, et al. [60] analyzed a debitcredit benchmark on a commercial database and found that all messages were less than 200 bytes. In the seven parallel scientific applications I studied in this chapter, I found that the average message size ranges between 19-230 bytes (Table 4.3).

Current microprocessors offer three mechanisms to effectively use the block transfer mechanism of memory buses. These are coalescing load/store buffers, block loads/stores, and cache blocks. A coalescing load/store buffer coalesces a processor's accesses to consecutive addresses (and, in some cases, the same address) and transfers them as a single block over the memory bus. Therefore, a processor's accesses to NI registers can be coalesced in the coalescing buffers and transferred as a single block.

Block load/store instructions—recently introduced in the Sun UltraSparc processor [119]—allows a processor to move a block of data between a device, such as main memory or NI, and the UltraSparc floating point registers. The Fujitsu AP3000 machine uses UltraSparc block load and store instructions to access the memory on the NI [109].

Finally, block transfer over the memory bus can be achieved by transferring data as cache blocks. However, this requires the NI to interact with the cache coherence signals, which are supported by most high-performance memory buses today. This is necessary to avoid having stale data in the processor's cache. Currently, most DMA-based NIs transfer data in coherent, cache block units over the memory bus to avoid this problem. Recently, Mukherjee, et al. [85] developed techniques using which processors and NIs can communicate more effectively via coherent, cache block transfers.

5.1.2 Degree of Processor Involvement for Transfer

Performance of data transfer depends not only on the size of the transfer, but also on how much the processor is involved in the transfer. Two design alternatives exist. The processor can initiate the transfer and allow the NI to manage the rest of the transfer. Alternatively, the processor itself can actively manage the transfer.[1] Each of these options have different design and performance implications. I discuss these options below.

5.1.2.1 NI manages transfer. If the NI manages the transfer, then the processor is usually required to only initiate the data transfer between the NI and the internal memory structures of a node. Currently, a processor can use one of three mechanisms to initiate rapid

1. A third option is possible in which a separate device or DMA engine manages the data transfer. I do not consider this option here.

data transfer to or from an NI: uncached load/store, User-Level DMA (UDMA), and cached store. An uncached load or store from the processor to a memory-mapped NI register can rapidly initiate data transfer from user space. However, an NI also needs physical memory addresses of data buffers from which it can obtain the data that must be transferred. Unfortunately, users cannot provide authenticated physical addresses of data buffers without violating most operating systems' protection model. Consequently, NIs must be prepared to fetch authentic physical addresses from the operating system [105, 47, 133].

To avoid the complexity of building an NI that fetches and manages authentic physical addresses, Blumrich, et al. [11, 88] proposed a low-overhead data transfer initiation scheme called User-Level DMA (UDMA). In this scheme users provide authentic physical addresses to the NI via a sequence of two user-level instructions: an uncached store and an uncached load. Additionally, UDMA allows users to directly deposit data into user data structures.

Unfortunately, the a key limitation of UDMA is that there is no known technique to extend UDMA in a general way to a multiprogrammed symmetric multiprocessing (SMP) node. The UDMA mechanism requires the two-instruction sequence to be atomic. However, in an SMP node, multiple such store-load sequences issued by multiple processors simultaneously can be overlapped leading to erroneous results. Markatos and Katevenis [75] showed the UDMA initiation sequence can be made atomic, but only under restricted conditions.

The multiprogramming problem faced by UDMA can be overcome using the third scheme in which processors and NIs communicate via cachable, shared memory. To send a message a processor simply writes to a location shared between the processor and the NI (e.g. increment the shared tail pointer of a shared queue). The NI polls the shared location to determine the presence of a message. Similarly, when a message arrives at the NI, the NI sets a shared location that the processor monitors. This scheme does not face the same multiprogramming problem of UDMA. This is because such an NI can directly read and write data to a portion of the user's address space, which is protected by the normal virtual memory mechanisms. However, like the first mechanism, this scheme does require the NI to fetch and manage authentic physical addresses to which the shared locations are mapped. Another drawback of this approach is that the NI must remember to poll the cached, shared locations to check for new messages. This is because, unlike uncached accesses, cached accesses by the processor is usually not visible outside the processor cache.

Cached stores additionally allow speculative processors to generate messages speculatively [87]. A processor's speculative stores are usually buffered locally inside the processor and committed only when the speculation succeeds. Consequently, a processor can speculatively issue a store to the cachable memory location shared between the processor and the NI. The store will, however, be visible (and the message committed) to the processor only after the speculation succeeds and commits.

5.1.2.2 Processor manages transfer. The previous subsection discusses solutions in which the processor initiates and the NI manages the data transfer. An alternative solution is to allow the processor to both initiate and manage the data transfer. For example, traditional program-controlled I/O requires direct processor involvement to transfer data between the processor and the NI. In this scheme a processor directly reads and writes data (instead of addresses) to memory-mapped NI registers via uncached loads and stores. Even the Ultrasparc block load and store instructions require processor involvement because these instructions block the processor until the data transfer is complete.

Processor-managed transfers usually simplify the NI design because an NI does not require authentic physical addresses to access a message. A processor's involvement for every data transfer, however, uses up precious processor resources, which can be used for other purposes (e.g. computation). Both UDMA and cache block transfers avoid processor involvement for data transfer, which reduces processor occupancy and allows overlap of computation with data transfer.

5.1.3 Source and Destination of Transfer

For both message send and reception data must be transferred between source and destination memories located in the processor, NI, or main memory. The source and destination of data transfer impact performance in two ways: determining what technology is used for the source and destination memories and whether or not data travels from the source to the destination directly.

Memory technology influences performance because the performance of current memories vary widely. DRAMs—the dominant technology used for main memory—is usually much slower than SRAMs, which are used to build processor memories, such as registers and caches. Consequently, transferring messages between the processor and NI via main memory, and not directly between the NI and processor, can hurt performance. Additionally, transferring messages between the processor and NI via main memory adds an extra hop over the memory bus, which adds extra latency. Nevertheless, if the NI memory overflows, it may be more useful to buffer messages in main memory rather than blocking the network or dropping the message. I discuss these issues in the next section.

5.2 Buffering Parameters

The amount of buffering available for an NI can have significant impact on an NI's performance (Section 2.3). Unfortunately, NIs cannot rely on network switches/routers to provide this level of buffering. Current commercial network switches/routers usually provide only a few hundred bytes of buffering (Table), which is usually sufficient to maintain the full bandwidth through the switch/router. However, if the receiving NI fails to remove messages from the network, the switches will block and send backpressure to the sender, thereby clogging up the network. Alternatively, switches, such as the Myricom Myrinet, simply drop messages if the receiving NI fails to eject the message from the network. For

Network Switch/Router	Maximum Buffering
Cray T3E router	105 bytes per non-adaptive virtual channel [106]
IBM Vulcan switch (SP2)	31 bytes + 1 Kbyte buffer pool shared between four ports [122]
Myricom M2M switch	20 bytes [38]
SGI Spider/Craylink switch	256 bytes per virtual channel [41]
TMC CM-5 network router	100 bytes [141]

Table 5.1: Buffering in commercial networks. This table shows the amount of buffering available between an input port and an output port in five commercial network switches/routers.

such networks either the NI must have sufficient buffering to rapidly remove messages from the network or software must guarantee reliable delivery, which incurs substantial overhead.

The rest of this section discusses two parameters that influence the amount of buffering available to an NI: where the NI buffers are located (Section 5.2.1) and how much the processor is involved to buffer messages. (Section 5.2.2).

5.2.1 Location of NI Buffers

The location of NI buffers is influenced by two goals that may often be conflicting. I want NI buffers to be located such that the processor can access them rapidly. However, I also want the NI buffers to be plentiful.

Allocating NI buffers in the NI itself allows direct and rapid data transfer between the NI and processor. Unfortunately, supporting large amounts of dedicated memory on the NI to buffer messages may not be economically feasible. In contrast, main memory can support large amounts of buffering, but may not allow rapid data transfers (Section 5.1.3). Traditionally, NIs have either allocated message buffers in dedicated NI memory, main memory, or a hybrid combination of the two. I discuss the implications of hybrid designs in the next subsection.

One compromise that allows the best of both is to distinguish between the logical and physical location of NI buffers. Logically, I can allocate the message buffers in coherent, shared memory, which is plentiful. Physically, however, such NI buffers can be located in processor caches, main memory, or NI memory. A host node's coherence protocol ensures that the different physical copies of the same (logical) message buffers are consistent across the node. In such a design, the NI memory behaves like another processor cache in an SMP node. Thus, in the common case, the processor can transfer data directly from the NI memory to the processor cache. However, if the NI memory overflows, the messages will be automatically replaced from NI memory to main memory, which allows plentiful buffering.

5.2.2 Degree of Processor Involvement for Buffering

If NI buffers are allocated both in dedicated NI memory and main memory, then either the processor or the NI must transfer messages from the dedicated NI memory to main memory. In the absence of such transfers, the network can fill up slowing down the entire system. More critically, in some situations, this can cause the system to deadlock. This is because the unavailability of message buffers can cause a cyclic dependence in which multiple processors are blocked (e.g. on a message send) waiting for other blocked processors to process incoming messages [68].

Transfer of messages from dedicated buffers to main memory can be managed by either the processor or the NI. Who (processor or NI) manages such transfers depends on how often such buffering is required. For NIs that always store message data to a node's main memory, processor involvement for buffering can seriously degrade performance. In contrast, NIs that are designed with the assumption that network "traffic jams" are rare occurrences may use processor-managed buffering as a fallback mechanism.

Network Interfaces	Simple description	Data Transfer Parameters						Buffering Parameters	
		Send			Receive				
		Size	Who manages transfer?	Source	Size	Who manages transfer?	Destination	Location	Proc. Involved?
NI_{2w}	TMC CM-5 NI-like [124]	Uncached	Proc.	Proc. Registers	Uncached	Proc.	Proc. Registers	NI / VM	Yes
NI_{64w}+Udma	Princeton Udma-based [11]	Block	NI	Cache/ Memory	Block	NI	Memory	NI / VM / Memory	Yes
NI_{16w}+Blkbuf	Fujitsu AP3000-like [109]	Block	Proc.	Block Buffer	Block	Proc.	Block Buffer	NI / VM	Yes
CNI_0Q_m	MIT StarT-JR-like [53]	Block	NI	Cache / Memory	Block	NI	Memory	Memory	No
$(NI_{16w}\text{+Blkbuf})_S$ $(CNI_0Q_m)_R$	DEC Memory Channel NI-like [44]	Block	Proc.	Block Buffer	Block	NI	Memory	Memory	No
$CNI_{512}Q$	Wisconsin CNI with no cache [85]	Block	NI	Cache / Memory	Block	NI	Proc. Cache	NI / VM	Yes
$CNI_{32}Q_m$	Wisconsin CNI with cache [85]	Block	NI	Cache / Memory	Block	NI	Proc. Cache	NI Cache / Memory	No

Table 5.2: Classification of memory bus NIs. This table classifies the seven memory bus NIs I evaluated in this chapter. Block denotes block transfer, Memory denotes main memory, Proc. denotes Processor, and VM denotes virtual memory. See Section 5.3 for an explanation of the taxonomy I use for NIs. NIs that involve the processor to manage data transfer between the NI and the processor have higher processor occupancy compared to the NIs that use NI-managed data transfers.

5.3 Network Interface Implementations

This section describes the seven NIs I evaluate in this chapter. Given the enormity of the design space exposed in Section 5.1 and Section 5.2, it would be hard to evaluate each and every component individually. Hence, I have selected seven NIs that, I believe, capture the essential components of the data transfer and buffering parameters. For all of my NIs, I assume a uniform network and flow control mechanism described in Section 5.4.

Table 5.2 lists the seven NIs I evaluate in this chapter. Column one uses the taxonomy I developed in Section 3.4 to describe the NIs. Column two gives a simple description of these NIs to aid readers in remembering which NI is which. I will use both descriptions (column one and two) interchangeably in the rest of the chapter.

NI_{2w} is a *CM-5-like NI* in which the processor can access only the first two words of the NI fifo. I study two variants of NI_{2w}. Section 5.4.2 compares a memory bus NI_{2w} with other memory bus NIs. Section 5.4.3 uses an NI_{2w}, which can be accessed in a single cycle, to approximate a processor-register-mapped NI. To distinguish this NI_{2w} from the memory bus NI_{2w}, I call it the single-cycle NI_{2w}.

NI_{64w}+Udma (Udma-based NI) allows the processor to examine the first 64 words of the NI fifo (256 bytes) and optionally transfer them to memory via the UDMA mechanism at both send and receive nodes (Section 5.1.2.1). Although the Udma-based NI implementation allows overlap of computation and data transfer, the messaging software waits until each UDMA transfer is complete. This reduces the complexity in the messaging software and avoids changes to the macrobenchmarks. This allows a uniform comparison across all seven NIs.

NI_{16w}+Blkbuf is an *AP3000-like NI*, which allows the processor to load and store 16 words (64 bytes) from the head of the fifo to a 64-byte send or receive block buffer located in the processor.[1] The processor accesses the block buffer via a load/store interface. These block buffers approximate the UltraSparc block load and store mechanism.

CNI_0Q_m is a *Start-JR-like NI* for which message queues reside in main memory. The ‘0’ in CNI_0Q_m indicates that CNI_0Q_m does not cache any message in the NI. CNI_0Q_m approximates the data transfer and buffering characteristics of the MIT StarT-JR NI [53]. However, unlike CNI_0Q_m, the StarT-JR NI resides on the I/O bus and does not use the lazy pointer and sense reverse optimizations.

1. The Fujitsu AP3000 NI has another mechanism to access the NI. For simplicity, I limit my discussion only to the way it accesses the NI via the processor’s block load/store instructions.

$(NI_{16w}+Blkbuf)_S(CNI_0Q_m)_R$ approximates the *Memory Channel NI* [44]. It denotes a hybrid NI in which the send interface resembles NI_{16w}+Blkbuf (AP3000-like NI) and the receive interface resembles CNI_0Q_m (the Start-JR-like NI). However, unlike the Digital Memory Channel NI, which attaches itself to the PCI I/O bus, I attach my Memory Channel-like NI directly to the memory bus to perform a uniform comparison with other NIs. Additionally, I do not use the multicast feature of the Memory Channel network because I focus specifically on its NI's data transfer and buffering parameters.

$CNI_{512}Q$ denotes a *CNI with no cache.* Its send and receive queues contain 512 64-byte blocks. $CNI_{32}Q_m$ is a *CNI with a cache*. That is, memory on the NI for both the send and receive queues is treated as 32-entry caches (with 64 byte cache blocks). See Section 4.1 for more details on these CNIs.

5.4 Results

This section examines the seven NIs' performance with respect to the data transfer and buffering parameters. All simulations in this section use the same methodology described in Section 4.2. Section 5.4.1 and Section 5.4.2 examine the performance of the sevens NIs with two microbenchmarks and seven macrobenchmarks respectively. Then, Section 5.4.3 compares the performance of the single-cycle NI_{2w} with $CNI_{32}Q_m$ (CNI with a cache), which performs the best for six of my seven macrobenchmarks and slightly worse than the AP3000-like NI for unstructured.

Throughout the rest of this section I will uniformly vary the number of network message buffers allocated at the sender and receiver. I call this parameter *flow control buffers*. So, for example, if the number of flow control buffers = 4 that implies that each NI has four outgoing buffers and four incoming network message buffers allocated for flow control.

5.4.1 Microbenchmarks

In this section I characterize the performance of seven NIs using two microbenchmarks: round-trip latency and bandwidth. These microbenchmarks capture the baseline performance of these NIs.

An alternative approach would be to characterize the NIs using the Berkeley LogP model [30]. The LogP model characterizes NI accesses with three parameters: latency (L), overhead or processor occupancy (o), and bandwidth (g). However, I refrain from using this model because the latency and overhead components of this model do not uniformly capture the same metrics for all of my NIs. For example, for coherent network interfaces, the latency component includes both the latency to transfer a message from the processor's cache to the NI and the network latency. In contrast, for CM-5-like NIs, the latency component captures only the network latency, while the actual data transfer is included in the overhead/occupancy component of the model. Nevertheless, the LogP model does help us understand qualitatively the performance of these NIs. For example, NIs that require processor involvement for data transfer have a higher processor occupancy compared to NIs that themselves manage the data transfer (Section 5.1.2.2).

5.4.1.1 Round-Trip Latency. Table 5.3 shows the *process-to-process* round-trip latency and bandwidth for my seven NIs. These numbers include the messaging layer overhead for copying a message from the NI to a user-level buffer and vice versa. Thus, for all NIs, except the Udma-based NI, data begins in the sending processor's cache and ends in the receiving processor's cache, rather than simply moving from memory to memory. Only for the Udma-based NI data begins in the sending processor's cache, but ends in the receiving processor's memory.

The round-trip latency numbers in Table 5.3 shows three important results. First, each of the three data transfer parameters—size of transfer, degree of processor involvement for transfer, and source/destination of transfer—have significant impact on the round-trip latency of each NI. Carefully choosing these parameters can improve the round-trip

Network Interface	**Round-Trip Latency**			**Bandwidth**			
	8	**64**	**256**	**8**	**64**	**256**	**4096**
CM-5-like NI	2.41	5.25	15.11	17	54	63	69
Udma-based NI	4.48	5.83	10.10	7	42	78	109
AP3000-like NI	1.95	2.48	4.47	26	154	234	298
Start-JR-like NI	1.54	2.38	5.04	29	119	191	221
Memory Channel-like NI	1.55	2.42	4.89	27	119	191	221
$CNI_{512}Q$	1.56	2.22	4.17	28	134	209	259
$CNI_{32}Q_m$	**1.29**	**1.78**	**3.42**	36	120	189	209
$CNI_{32}Q_m$+Throttle	n/a	n/a	n/a	**36**	**158**	**272**	**351**

Table 5.3: Process-to-process round-trip latency for seven NIs. This table shows the process-to-process round-trip latency (in microseconds) for 8-, 64-, and 256-byte message payload and process-to-process bandwidth (megabytes per second) for 8-, 64-, 256-, and 4096-byte message payload. Each message contains an eight-byte header. $CNI_{32}Q_m$+Throttle throttles the sender to match the maximum message consumption rate of the receiving NI. Send throttling does not significantly change the bandwidth attained by any other NI. For all these numbers, I set the number of flow control buffers = 8.

latency by more than a factor of three. Second, the relative importance of these parameters depend on the specific NI design. Third, among the seven NIs, $CNI_{32}Q_m$—the CNI with a cache—offers the best round-trip latency because it optimizes the three data transfer parameters.

Below I examine five interesting comparisons revealed by Table 5.3:

The Udma-based NI performs better than CM-5-like NI only for messages greater than 96 bytes

The Udma-based NI's round-trip latency is worse than CM-5-like NI for messages with payload less than 96 bytes (the exact breakeven point is not shown in the table), but substantially better as the message payload increases beyond this size. This is because for small messages, the Udma-based NI's high initiation overhead (one uncached store + one uncached load + switch bus master from processor to NI) offsets its two advantages: ability to transfer messages in blocks and ability to directly deposit data in user space without processor involvement. Hence, for my macrobenchmarks the Udma-based NI attempts to use the UDMA mechanism only for messages with payload greater than 96 bytes.

The AP3000-like performs substantially better than the Udma-based NI

The AP3000-like NI performs substantially better (more than a factor of two) than the Udma-based NI, even though all transfers are managed by processor for the AP3000-like NI. This is because like the Udma-based NI, it transfers messages in blocks, but unlike the Udma-based NI, it has a low initiation overhead (an uncached store) and it transfers data directly to the fast receive block buffer residing on the processor chip (and not into slower main memory).

The Start-JR-like NI and AP3000-like NI have a crossover point

The Start-JR-like NI outperforms the AP3000-like NI for messages less than 64 bytes (size of the block buffer) because the AP3000-like NI has higher overhead (12 processor cycles) to flush and load the send and receive block buffers respectively. Beyond a 64-byte message payload, the AP3000-like NI's overhead is amortized and consequently it outperforms the Start-JR-like NI. The Memory Channel-like NI's round-trip latency is almost similar to that of the Start-JR-like NI, which indicates that the send side of the Start-JR-like NI and the Memory Channel-like NI exhibit almost similar performance.

$CNI_{512}Q$ outperforms the Start-JR-like NI

$CNI_{512}Q$—the CNI with no cache and queues allocated in dedicated CNI memory—outperforms Start-JR-like NI, even though the memory on $CNI_{512}Q$ is as slow as main memory. The difference arises because of two reasons. First, on the receive side messages are steered to processor caches directly from the NI and not via main memory, which adds additional latency for the Start-JR-like NI. Second, on the send side, for messages larger than a cache block (i.e., 64 bytes), $CNI_{512}Q$ prefetches cache blocks as the processor composes them in its cache. For example, while a processor is composing a cache block of a message, $CNI_{512}Q$ fetches the previous block of the same message. This fetch is initiated by $CNI_{512}Q$ when it observes the processor's request for exclusive access for a subsequent cache block of a message. If a cache block is fetched too early even before entire cache block is written, then $CNI_{512}Q$ can re-fetch it when the message is ready for delivery. Thus, avoiding processor involvement for data transfer allows simultaneous transfer and creation of a message. Unlike $CNI_{512}Q$, the Start-JR-like NI cannot prefetch cache blocks of a message because it does respond to the memory coherence signals (e.g. coherent invalidations).

$CNI_{32}Q_m$ shows the best round-trip latency

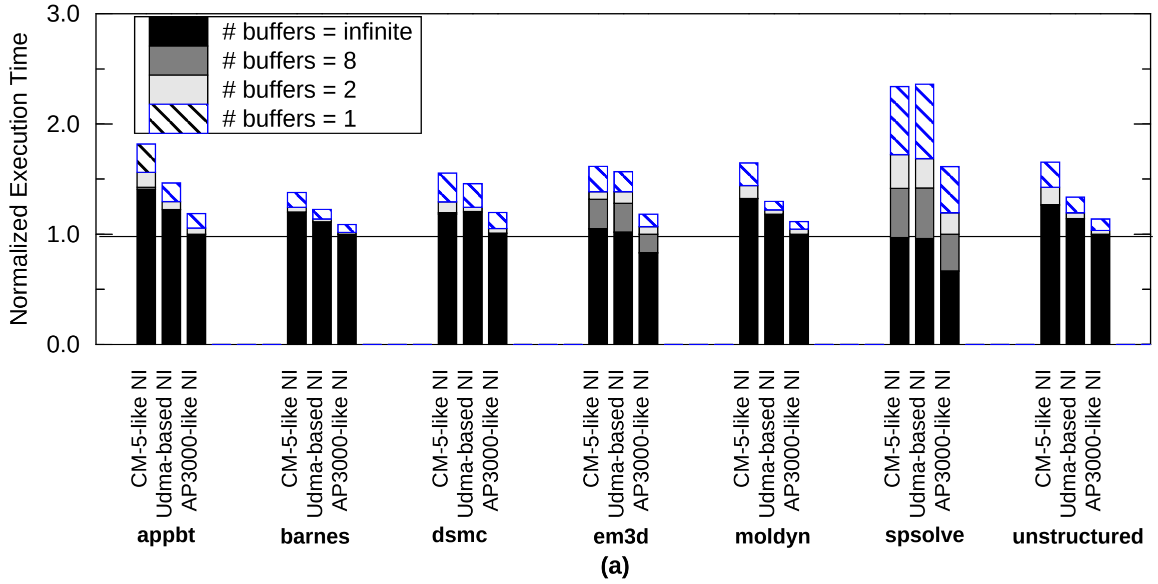
$CNI_{32}Q_m$—the CNI with a cache—shows the best round-trip latency among my seven NIs because it provides all the benefits of $CNI_{512}Q$, but with smaller and faster cache memories compared to $CNI_{512}Q$. Therefore, overall it outperforms all other NIs by roughly 20%-342% for message payload between 8-256 bytes.

In summary, I find that low latency transfer can be achieved for small messages via block transfers, minimal processor involvement, and direct processor-to-NI transfers. The relative importance of these parameters depend on the specific NI designs.

5.4.1.2 Bandwidth. The bandwidth numbers in Table 5.3 show trends similar to the round-trip latency numbers with two key exceptions that I discuss below.

The AP3000-like NI offers significantly greater bandwidth compared to the Start-JR-like NI and $CNI_{512}Q$

This is because at the receive side the AP3000-like NI transfers messages directly from the small and fast NI memory to the receive block buffer located next to the processor. This is significantly faster than reading messages from the slower main memory for the Start-JR-like NI and slower NI memory for $CNI_{512}Q$.



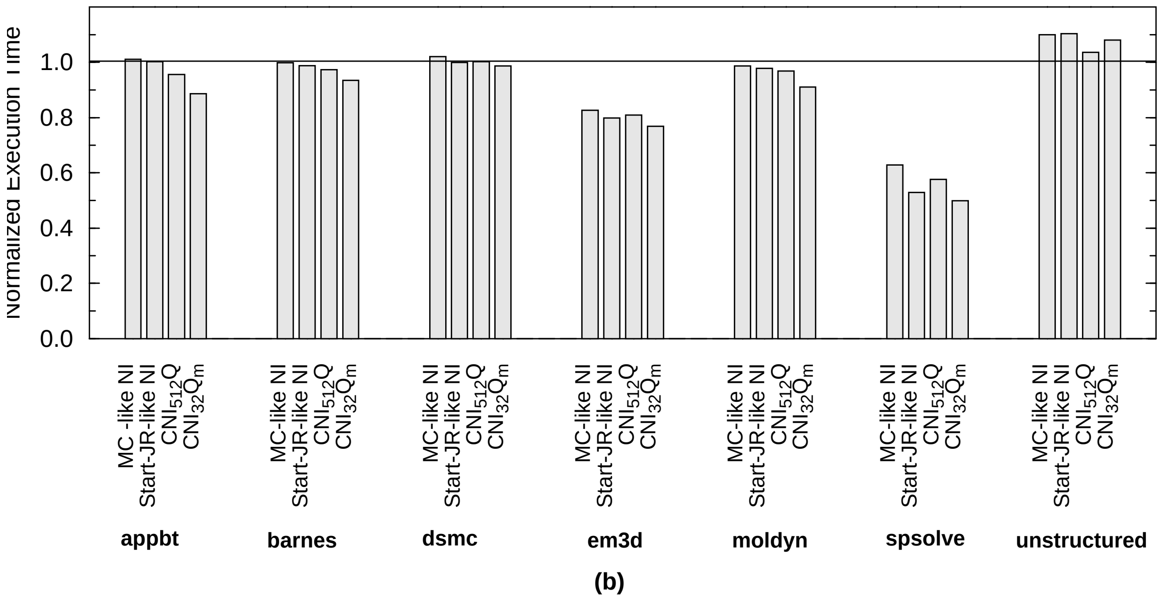


Figure 5-2. Comparison of execution time of seven NIs. Both (a) and (b) are normalized to the AP3000-like NI with number of flow control buffers = 8. # buffers denotes the number of flow control buffers. (a) compares the performance of my three fifo-based NIs for different levels of flow control buffering. The black shade represents the execution time with infinite flow control buffering. Lighter shades represent the incremental execution time penalty for three flow control buffering levels (8, 2, and 1). (b) compares the performance of four coherent network interfaces with number of flow control buffers = 2. I However, because these NIs themselves provide plentiful buffering, their performance is largely insensitive to the number of flow control buffers. MC-like NI denotes the Memory Channel-like NI.

Without throttling $CNI_{32}Q_m$'s bandwidth is worse than the AP3000-like NI, even though $CNI_{32}Q_m$'s latency is significantly better.

This is because $CNI_{32}Q_m$'s send bandwidth is significantly greater than its receive bandwidth. This causes $CNI_{32}Q_m$'s receive cache to overflow, which forces the receiving processor to pick up most messages from main memory, like the Start-JR-like NI. However, appropriately throttling the sending processor after every send can help improve $CNI_{32}Q_m$'s bandwidth by preventing the receive cache from overflowing. This allows the receiving processor to pick up messages from the fast $CNI_{32}Q_m$ memory, instead of slower main memory. However, I do not see this effect in my macrobenchmarks.

Overall, with send throttling, $CNI_{32}Q_m$ achieves a bandwidth of 351 megabytes/second, which is significantly greater than the bandwidth offered by any other NI.

5.4.2 Macrobenchmarks

This section discusses the performance of my seven NIs with seven macrobenchmarks (Section 4.2.3). Overall, I can draw two conclusions. First, the data transfer parameters have significant impact on the performance of all seven macrobenchmarks. Second, buffering only affects two of the macrobenchmarks: em3d and spsolve. This is because both these benchmarks generate small messages more rapidly than the receiving processor can consume. Consequently, for em3d and spsolve, buffering is more important than data transfer.

I do not, however, attempt to quantify the relative importance of each of the three data transfer and two buffering parameters. This is because the extent to which each parameter affects a macrobenchmark depends on the specific NI design and the macrobenchmark itself. Nevertheless, it should be noted that each of these parameters directly affects performance. This is because for pure shared-memory applications, such as appbt and barnes, which communicate using a request-response protocol, all the parameters adds latency to the requests and responses. For pure message-passing applications, such as em3d and spsolve, the receive side is the bottleneck. Consequently, all the parameters on the receive side is on the critical path. The rest of the applications that use hybrid protocols—that is, both message passing and shared memory—consequently have the same behavior.

I divide my discussion into two parts and examine the results in detail. First, I discuss the performance of the CM-5-like NI, the Udma-based NI, and the AP3000-like NI (Section 5.4.2.1). These NIs rely on NI memory to buffer network messages (via the flow control buffers). These three NIs are fifo-based NIs, but differ in the way they pop/push data to the fifos. For such push and pop the CM-5-like NI uses uncached loads/stores, the Udma-based NI uses User-Level DMA (or UDMA), and AP3000-like uses block loads/stores.

Second, I discuss the Memory Channel-like NI, the Start-JR-like NI, $CNI_{512}Q$, and $CNI_{32}Q_m$, (Section 5.4.2.2). These NIs provide plentiful buffering in main memory without requiring a processor's involvement. All these four NIs are either fully coherent or partially coherent. They differ primarily in the way the NI queues are allocated. The Memory Channel-like NI allocates receive queues in main memory. The Start-JR-like NI allocates both send and receive queues in main memory. $CNI_{512}Q$ allocates the queues in dedicated CNI memory. Finally, $CNI_{32}Q_m$ allocates queues in main memory, but caches them in a CNI cache.

5.4.2.1 Comparison of Three Fifo-based NIs. Figure 5-2a compares the execution time for the three fifo-based NIs. The black bars, which show the execution time for the three NIs for infinite flow control buffering, allow us to isolate the impact of the data transfer parameters on the macrobenchmarks. With infinite flow control buffering, the Udma-based NI outperforms the CM-5-like NI by 0-15% and the AP3000-like NI outperforms the Udma-based NI by 11-44%. The Udma-based NI performs similar to or better than the CM-5-like NI because it uses the UDMA mechanism only for large payloads and falls back on uncached loads and stores, like the CM-5-like NI, for smaller messages. The AP3000-like NI's lower latency and greater bandwidth (Section 5.4.1) clearly help improve the macrobenchmarks' performance.

The lighter bars in Figure 5-2a show the increase in execution time as I reduce the number of flow control buffers. Clearly, the number of flow control buffers have a significant impact on performance. Figure 5-2a shows two interesting results about flow control buffering. First, for all three NIs and all of our seven applications, increasing the number of flow control buffers from one to two significantly improves performance (between 6-40%). However, increasing the number of flow control buffers beyond two buys only modest performance gains (less than 19%) for most applications, except em3d and spsolve.

Second, the number of flow control buffers has significant impact on em3d and spsolve. This is because both em3d and spsolve generate bursts of small messages (less than 20 bytes) more rapidly than the receiving NI can consume. Consequently, the lack of flow control buffers has a dramatic impact on performance. For em3d and spsolve increasing the number of flow control buffers from two to infinity improves performance by 29-40% and 78-101% respectively for the three NIs. Actually, increasing the number of flow control buffers to 128 for em3d and 33 for spsolve captures most of the performance gains that can be achieved from an infinite number of flow control buffers.

5.4.2.2 Comparison of Four Coherent Network Interfaces. Figure 5-2b compares the execution time (normalized to the AP3000-like NI for flow control buffers = 8) of four NIs that are either partially or fully coherent. These NIs provide NI-managed, plentiful buffering in main memory on the receive side. Consequently, these NIs are largely insensitive to the number of flow control buffers.

Figure 5-2b shows three interesting results. First, the performance of the Memory Channel-like NI varies widely for the seven macrobenchmarks. It performs significantly better than the AP3000-like NI, with the number of flow control buffers = 8, for em3d and spsolve because it provides plentiful buffering in main memory without direct processor involvement. It performs almost similar to the AP3000-like NI for appbt, barnes, dsmc, and moldyn because these macrobenchmarks do not gain significantly from plentiful buffering. It performs worse than the AP3000-like NI for unstructured because unstructured's large messages effectively use the greater bandwidth provided by the AP3000-like NI (Table 5.3).

Second, among the four NIs shown in Figure 5-2b, the Memory Channel-like NI performs the worst and $CNI_{32}Q_m$ performs the best. $CNI_{32}Q_m$—the CNI with a cache—outperforms the Memory Channel-like NI by 2-26% for the seven macrobenchmarks due to its lower latency and higher bandwidth (Table 5.3). $CNI_{32}Q_m$ also outperforms The AP3000-like NI—the best fifo-based NI—and $CNI_{512}Q$—the CNI with queues allocated in main memory—for all applications, except unstructured. It appears that the most important feature of unstructured’s communication is to stream data from the sender to the receiver. Both the AP3000-like NI and $CNI_{512}Q$ has less overhead for this data streaming compared to $CNI_{32}Q_m$, which incurs extra overhead due to its cache management (e.g. cache replacement). Consequently, $CNI_{32}Q_m$ performs marginally worse than the AP3000-like NI and $CNI_{512}Q$ for unstructured. Like Mukherjee, et al. [85], I find that $CNI_{32}Q_m$ is competitive with $CNI_{512}Q$ with much less memory.

Third, a comparison of the Start-JR-like NI and $CNI_{32}Q_m$ shows that caching messages in a CNI cache, as in $CNI_{32}Q_m$, provides a performance boost of 2-13% for the seven macrobenchmarks. An examination of NI-related memory bus transactions reveals that $CNI_{32}Q_m$ reduces the number of main memory to processor cache transactions by 54% (averaged across the seven macrobenchmarks). This is because $CNI_{32}Q_m$ provides messages to the processor via direct CNI-cache-to-processor-cache transfers. Further, as the performance gap between microprocessors and main memory widens, I expect $CNI_{32}Q_m$ to provide significantly better performance than the Start-JR-like NI because of two reasons. First, because $CNI_{32}Q_m$ caches are small, they can be built with faster SRAMs, thereby providing lower latency to transfer messages. Second, $CNI_{32}Q_m$ satisfies more than 50% of the processor’s accesses to the NI directly from its cache, which avoids message steering via main memory.

Overall, I find that $CNI_{32}Q_m$—the coherent interface with a cache—performs the best because it optimizes all of the five data transfer and buffering parameters. In summary, $CNI_{32}Q_m$:

- effectively uses the block transfer mechanism of current memory buses by transferring messages in cache blocks;
- minimizes processor involvement for data transfer by initiating the transfer using a cachable store and decoupling the processor and NI via memory-mapped, cachable queues;
- directly transfers messages from the NI cache to the processor cache in the common case;
- provides plentiful buffering in main memory; and
- allows the NI to directly deposit messages into main memory, when the NI cache overflows.

5.4.3 Single-Cycle NI_{2w} vs. $CNI_{32}Q_m$

Figure 5-3 compares the performance of $CNI_{32}Q_m$ with an NI_{2w} NI, whose memory can be accessed by the processor in a single cycle. Thus, my single-cycle NI_{2w} approximates processor-register-mapped NIs in research machines, such as the MIT M-machine [39].[1]

Figure 5-3 shows two interesting results. First, $CNI_{32}Q_m$—the CNI with a cache—outperforms my single-cycle NI_{2w} for spsolve and em3d for small number of flow control buffers. Processor-register-mapped NIs are likely to have a small number of flow control buffers because of two reasons. First, a processor's register memory is a precious resource and its size is severely constrained by its access time. Second, the demands of multiprogramming require that the NI_{2w} buffers be either partitioned among multiple processes or saved and restored across context switches. The first solution limits the number of flow control buffers allocated per process and the second solution increases the context-switch

1. Unlike my single-cycle NI_{2w}, a processor in the MIT M-machine can compute directly from the NI registers, which allows zero-cycle access to the NI registers for some cases.

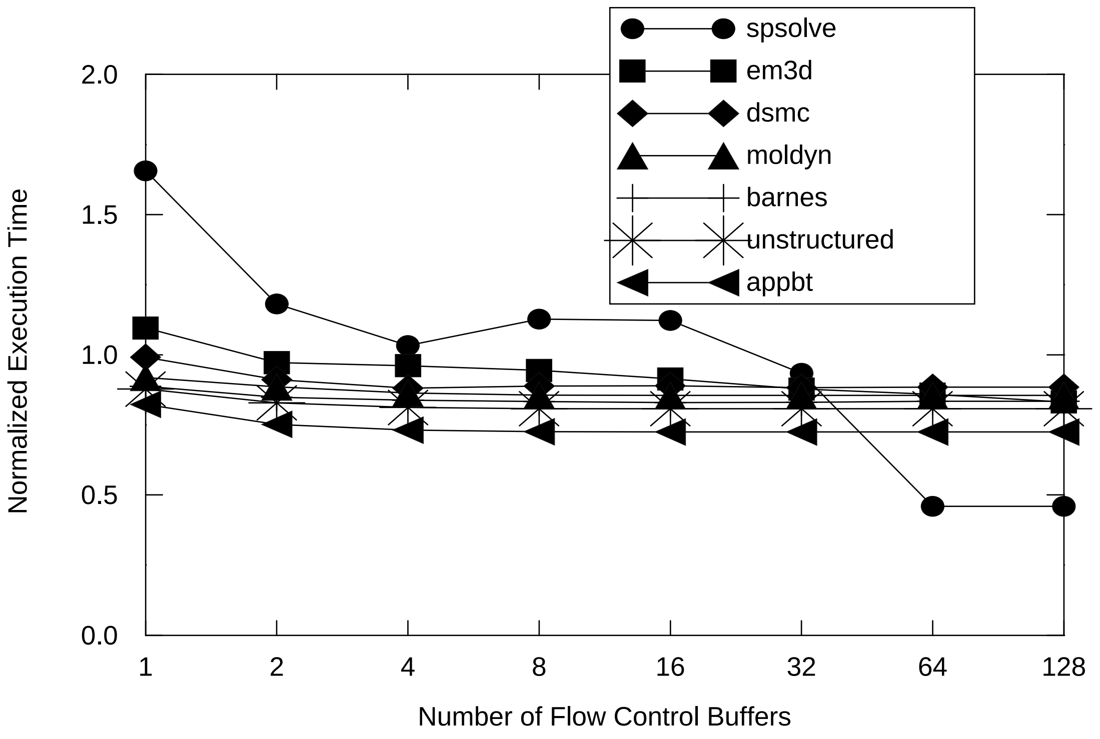


Figure 5-3. Comparison of execution of a single-cycle NI_{2w} with $CNI_{32}Q_m$. The vertical axis is normalized to the $CNI_{32}Q_m$ on the memory bus. $CNI_{32}Q_m$ is independent of flow control buffering because it provides plentiful buffering in main memory.

time. Further, my single-cycle NI_{2w} cannot also rely on commercial NIs for plentiful buffering (see Table). Consequently, $CNI_{32}Q_m$'s ability to buffer messages in NI caches and main memory without processor involvement makes its performance better or comparable to the single-cycle NI_{2w} for spsolve and em3d. For example, for flow control buffers = 2, $CNI_{32}Q_m$'s performance is better than the single-cycle NI_{2w} by 18% for spsolve and comparable for em3d. For spsolve and em3d, the breakeven point between $CNI_{32}Q_m$ and the single-cycle NI_{2w} occurs when the number flow control buffers equals 32 and 2 respectively.

Second, for the five macrobenchmarks other than spsolve and em3d, $CNI_{32}Q_m$ is within 15% of the performance of the single-cycle NI_{2w} (averaged across the five macrobenchmarks).

The above results suggest that in the absence of adequate buffering, mapping an NI directly to the processor registers may not always be the optimal design point. Perhaps a two-level register memory hierarchy for NI registers can make such processor-register-mapped NIs competitive with a memory bus NI, such as $CNI_{32}Q_m$.

5.5 Related Work

To the best of my knowledge, this work is the first to systematically identify, examine, and explore the data transfer and buffering parameters that underlie the design of high-performance NIs for fine-grain communication. Karamcheti and Chien [58] compared the messaging support in TMC CM-5 and Cray T3D and concluded that requiring processor involvement for message reception can significantly degrade performance. I improve upon their work by exposing and examining the design space of data transfer and buffering parameters. Blumrich, et al. [14] compared the SHRIMP I and SHRIMP II NIs, but did not explore alternate data transfer and buffering mechanisms. Mackenzie, et al. [73] studied the effect of buffering using a synthetic workload and concluded that buffering messages in virtual memory can occur only rarely for realistic applications. However, in contrast I found that for two of my seven macrobenchmarks, buffering can play a significant role in improving performance. Henry and Joerg [50] compared the performance of three NIs mapped respectively to the processor registers, L1 cache bus, and an off-chip L2 cache bus. However, unlike my study, they did not examine the impact of buffering on the performance of these NIs.

5.6 Conclusions

In this chapter I have systematically identified, examined, and explored two key parameters—data transfer and buffering—that affect the design of high-performance NIs targeted for fine-grain communication. The data transfer parameters capture how messages are transferred between internal memory structures (e.g. processor caches, main memory) of a computer and a memory bus NI. The buffering parameters capture where and how an NI buffers incoming network messages. I found that each of the three data transfer parameters—size of transfer, degree of processor involvement for transfer, and source/destination of transfer—and two buffering parameters—location of buffers and degree of processor involvement for buffering—can have a significant impact on performance.

Using two microbenchmarks and seven macrobenchmarks I evaluated seven memory bus NIs that I believe captured the essential components of the design space exposed by the five data transfer and buffering parameters. These seven NIs abstract the data transfer and buffering parameters of the NIs in TMC CM-5, Fujitsu AP3000, Princeton User-Level DMA, Digital Memory Channel, MIT StarT-JR, and two Coherent Network Interfaces—$CNI_{512}Q$ and $CNI_{32}Q_m$—proposed in this thesis.

Overall, I found that among these seven NIs, $CNI_{32}Q_m$—a coherent network interface that treats memory on the interface as a cache—performed the best because it optimizes all five data transfer and buffering parameters. It:

- effectively uses the block transfer mechanism of current memory buses by transferring messages in cache blocks,
- minimizes processor involvement for data transfer by initiating the transfer using a cachable store and decoupling the processor and NI via memory-mapped, cachable queues,
- directly transfers messages from the NI cache to the processor cache in the common case,
- provides plentiful buffering in main memory, and
- allows the NI to directly deposit messages into main memory, when the NI cache overflows.

As a corollary of this study, I found that, contrary to conventional wisdom, mapping an NI to the processor registers may not always be the ideal choice. This is because processor register memory is a precious resource, which may not provide adequate buffering for some applications. Consequently, for two of my seven macrobenchmarks, I found that $CNI_{32}Q_m$ outperformed a processor-register-mapped NI with small amounts of buffering.

Chapter 6

Using Prediction to Accelerate Coherence Protocols

Chapters 2 - 5 explored techniques to accelerate the performance of network interfaces for system area networks. These techniques can accelerate user-to-user messaging in a parallel machine programmed with explicit message-passing. In this chapter I examine techniques to accelerate the communication performance of parallel machines programmed with a shared-memory programming model.

Shared-memory communication interfaces differ from network interfaces in at least two ways. First, shared-memory communication interfaces allow processors to access memory using a single address space, even though some memories may be located in a remote computer (Figure 1-3).

Second, interaction of processor and shared-memory interfaces is usually much faster than processor-NI interactions. This is because shared-memory communication interfaces usually hardwire protocols that prepare messages in hardware or firmware to fetch remote memory blocks. In contrast, network interfaces typically serve as a conduit for messages generated by processors.

Unfortunately, performance problems with shared-memory interfaces arise in the hardwired protocol itself. Modern shared-memory interfaces in large, shared-memory multiprocessors allow processors to transparently cache remote memory. Such interfaces use a form of cache coherence protocol called a directory protocol (Section 6.1.1) to keep per-processor caches coherent. However, a hardwired directory protocol may not match an application's shared-memory communication patterns (also known as *sharing patterns*). Consequently, memory references to remotely-cached blocks that invoke the directory protocol can suffer long latencies. To ameliorate this latency, researchers have augmented standard coherence protocols with optimizations for specific sharing patterns, such as read-modify-write, producer-consumer, and migratory sharing. This chapter seeks to replace these directed solutions with general prediction logic that monitors coherence activity and triggers appropriate coherence actions.

The first contribution of this chapter is the design of the *Cosmos* coherence message predictor for accelerating coherence protocols (Section 6.2). Cosmos' design is inspired by Yeh and Patt's two-level *PAp* branch predictor [139] (Section 6.1.2). Cosmos makes a prediction in two steps. First, it uses a cache block `address` to index into a *Message History Table* to obtain one or more `<processor,message-type>` tuples. These `<processor,message-type>` tuples correspond to sender and message type of the last few coherence messages received for that cache block. Second, it uses these `<processor,message-type>` tuples to index a *Pattern History Table* to obtain a `<processor,message-type>` prediction. Notably, Cosmos faces a greater challenge than branch predictors because the Cosmos' prediction is a multi-bit `<processor,message-type>` tuple rather than a single bit branch outcome.

This chapter concentrates on coherence protocol message prediction in isolation (analogous to studying branch prediction in isolation). I do not integrate the Cosmos predictor into a coherence protocol for two reasons. First and most important, my tools are not ready to handle a full timing simulation of a protocol that can be accelerated using prediction. Second, I do not want initial results in this area obscured by implementation idiosyncrasies. Nevertheless, I expect such integration to be successful because the integration of directed predictions has been successful [66,67,28, 120]. Section 6.3 briefly discusses possibilities for such integration.

The second contribution of this chapter is a detailed evaluation of the Cosmos coherence message predictor. Section 6.4 states methodological assumptions, including the use of five scientific benchmarks on a target shared-memory machine with 16 processors running the Stache directory protocol [100]. Section 6.5 gives Cosmos' prediction rates and analyzes application details. Variations of Cosmos predict the source and type of the next coherence message with surprisingly-high accuracies of 62-69% (*barnes*), 84-86% (*moldyn*), 84-85% (*appbt*), 74-92% (*unstructured*), and 84-93% (*dsmc*). Cosmos' high prediction accuracy results from predictable coherence message patterns or *signatures* associated with specific cache block addresses. Such signatures are generated by sharing patterns [9, 46] that do not change or change very slowly during the execution of these applications. Cosmos' lower accuracy for *barnes* occurs because *barnes* periodically re-builds its principal data structure (an octree), thereby moving logical nodes (with stable sharing patterns) to different memory addresses (obscuring sharing patterns from Cosmos).

Section 6.7 explores the implications of Cosmos. Clearly, coherence message prediction works, because sharing patterns are often stable. Others have exploited sharing patterns with directed optimizations, such as dynamic self-invalidation and migratory protocols. Using Cosmos could be better (or worse) than directed predictors due to performance and implementation issues. Cosmos can perform better, because it can discover and track application-specific patterns not known *a priori* (e.g., as occurs for *unstructured*). It can perform worse if it is slower to recognize known patterns. Cosmos' implementation complexity can be less, because predictor logic is separated from the standard protocol logic (unlike previous directed predictors that are intertwined with the standard coherence protocol). Cosmos, however, is likely to require more state than directed optimizations. In summary (Section 6.8), Cosmos' high prediction accuracies justify more investigation into using prediction to accelerate coherence protocols.

6.1 Background

This section describes the structure of a basic directory protocol (Section 6.1.1) and reviews Yeh and Patt's two-level adaptive branch predictor (Section 6.1.2). In the next section I discuss how Cosmos—a modified version of Yeh and Patt's two-level predictor—can predict a directory protocol's messages with high accuracy. Throughout the rest of the chapter I will use the terms "node" and "processor" interchangeably because I consider only single-processor nodes to simplify my discussion.

6.1.1 Structure of a Directory Protocol

Most large-scale shared-memory multiprocessors use a directory protocol to keep multiple caches coherent. A directory protocol associates state with both caches and memory. This state is typically maintained at a cache block (e.g. 32-128 bytes) granularity. The state associated with each memory block is referred to as a directory entry.

The directory entry for each memory block records whether or not a memory block is idle (that is, no cached copies exist), a writable copy of the block exists, or one or more readable copies of the block exist. To simplify the discussion I only consider a full-map and write-invalidate directory protocol, such as the SGI Origin protocol [69]. A directory entry in such a protocol maintains logical pointers to all caches that hold a valid copy of the block and invalidates all outstanding copies of the block when one processor wishes to write to it. Similarly, a block in a cache is usually in one of three quiescent states: invalid, shared, or exclusive. These states define whether a processor's load or store can access the cache block. Processors must involve coherence actions on loads to invalid blocks and on stores to shared (i.e. read-only) and invalid blocks.

A cache coherence protocol can, therefore, be viewed simply as a finite-state machine that changes state in response to processor accesses and external messages. For caches state transitions occur in response to processor accesses and messages from the directory (and possibly other caches). A directory entry changes state in response to messages from

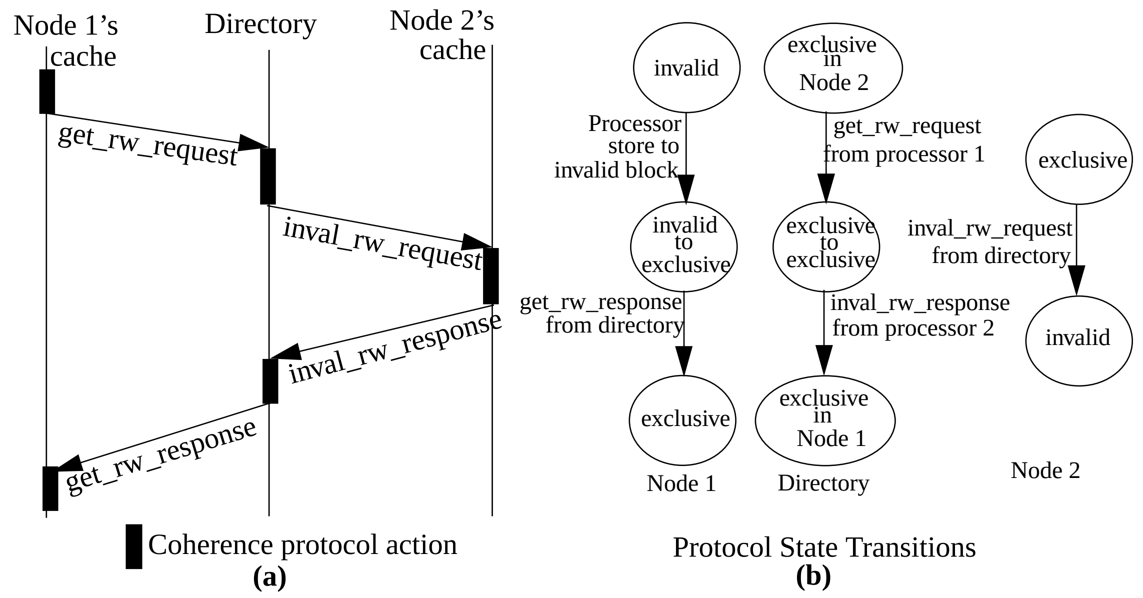


Figure 6-1. Basic structure of a directory protocol. (a) shows message exchange between a directory and two caches and (b) shows the corresponding state transitions. Table 6.1 contains an explanation of the coherence message types. Initially, processor 2 has an exclusive copy of a cache block. Processor 1 issues a store to the block. This invokes the coherence protocol, which sends a message to the directory. The directory examines its state and sends a message to processor 2 requesting it to return the block to the directory and invalidate its copy of the block. When it receives the block from processor 2, it forwards it to processor 1, which marks the cache block as exclusive in processor 1. The states "invalid to exclusive" and "exclusive to exclusive" represent transition states.

caches. Figure 6-1 shows an example of message exchange and state transitions in two caches and a directory.

Unfortunately, the finite-state machine that implements the coherence logic often incurs multiple long-latency operations. These latencies can become severe if coherence actions are implemented in software [104, 100, 80] or firmware [71]. Additionally, a directory may need to exchange messages with other caches before it can respond to a processor's request for a memory block. Such message exchange can also introduce substantial delay in the critical path of a remote access. For example, Figure 6-1a shows that a processor's store to a block that resides in another node's cache may require as many as five coherence protocol actions at different caches and the directory and four message traversals across the network that connects these caches and the directory. In some protocols, such as the SGI Origin and Stanford DASH protocols, node 2's cache can forward the response directly to node 1. However, this only reduces the critical path of a remote access to four coherence actions and three coherence messages.

6.1.2 Two-Level Adaptive Branch Predictor

A branch predictor predicts whether the branch will be taken or not taken. Correct prediction of branch directions improves the performance of wide-issue, deeply pipelined microprocessors because it allows them to fetch and execute probable instructions without waiting for the outcome of previous branches. J. Smith [114] proposed several dynamic branch predictors that use program feedback to increase the accuracy of branch prediction. More recently, Yeh and Patt proposed a general dynamic branch predictor called *PAp* [139]. PAp is a two-level adaptive predictor that makes a prediction for a branch based on the sequence of branches a program executed before it arrived at the particular branch. PAp makes a prediction in two steps. First, it uses the program counter of a branch to index into a *Branch History Table* to obtain k bits, which represent the outcomes of the last k branches at this program counter. Second, it uses these k bits to index a Per-Branch *Pattern History Table* to obtain a prediction. Each entry in the Pattern History Table is a finite-state machine, which returns predictions based on the behavior of a finite number of previous occurrences of this branch (and the k bits from the Branch History Table). In the next section I will show how PAp can be modified to obtain coherence message predictions.

6.2 Predicting Coherence Protocol Messages

In this section I will study the Cosmos coherence message predictor. In the next section I will briefly describe how Cosmos can accelerate coherence protocols. This section begins with an example of a producer-consumer sharing pattern and its corresponding coherence message signature. The rest of the section uses this example to describe Cosmos in detail.

Messages Received by Directory from Caches		**Messages Received by a Cache from a Directory**	
Message	**Description**	**Message**	**Description**
get_ro_request	get block in read-only (shared) state	get_ro_response	response to get_ro_request
get_rw_request	get block in read-write (exclusive) state	get_rw_response	response to get_rw_request
upgrade_request	upgrade block from read-only to read-write	upgrade_response	response to upgrade_request
inval_ro_response	response to inval_ro_request	inval_ro_request	invalidate read-only (shared) copy of block
inval_rw_response	response to inval_rw_request	inval_rw_request	invalidate read-write (exclusive) copy and return block

Table 6.1: Sample of coherence messages. A sample of coherence messages usually found in full-map, write-invalidate coherence protocols.

6.2.1 Signature Generated by Producer-Consumer Sharing Pattern

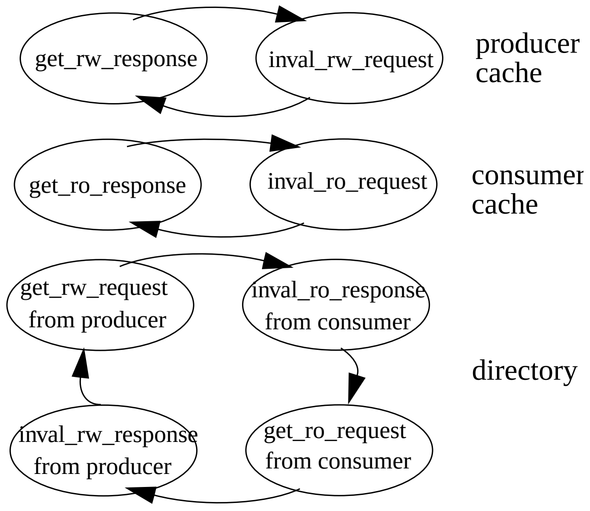
Figure 6-2 shows an example of a producer-consumer sharing pattern and how it can lead to predictable message patterns or *signatures* for a particular cache block. For example, the producer in Figure 6-2 observes the following message sequence:

send `get_rw_request` to directory
receive `get_rw_response` from directory
receive `inval_rw_request` from directory
send `inval_rw_response` to directory

Figure 6-2b shows the incoming message signature that results from the above message sequence. Figure 6-2b, however, represents a simple case. Consider a slightly more complex example in which the pseudo code in Figure 6-2a is extended to support two consumers instead of one. In this case the producer and the two consumers will still follow the same predictable signatures as shown in Figure 6-2b. However, at the directory the two `get_ro_request` messages can now arrive in any order from the two consumers. But, the arrival of a `get_ro_request` from the first consumer suggests strongly the possibility of the arrival of another `get_ro_request` from the second consumer and vice versa. To achieve high accuracy a predictor must adapt to such variations in the incoming message

```
/* private_counter = private variable */
/* shared_counter = shared variable */
repeat
  ...
  if (producer)
    private_counter++
    shared_counter = private_counter
    barrier
  else if (consumer)
    barrier
    private_counter = shared_counter
  else
    barrier
  endif
  ...
until done
```

(a)


get_rw_response
inval_rw_request
producer cache
get_ro_response
inval_ro_request
consumer cache
get_rw_request from producer
inval_ro_response from consumer
directory
inval_rw_response from producer
get_ro_request from consumer

(b)

Figure 6-2. Message signature generated by a producer-consumer sharing pattern. (a) shows a pseudo code for the producer-consumer sharing pattern. A producer writes to a shared counter and a consumer reads the shared counter. (b) shows the sequence of messages received by the producer cache, consumer cache, and directory for the cache block containing the shared counter (assuming no false sharing). Table 6.1 explains the different message types shown in this figure.

stream. The rest of this section discusses the design of such an adaptive predictor called Cosmos.

6.2.2 Basic Structure of Cosmos

The previous subsection suggests that a coherence message predictor must adapt to an incoming coherence message stream based on two properties:

- address of cache blocks, because sharing patterns of different cache blocks may differ, and
- history of messages for a cache block, because a stream of incoming coherence messages correspond to fixed sharing patterns for specific cache blocks. [1]

1. We will see in Section 6.5.1 that a history of three messages achieves most of the prediction accuracy.

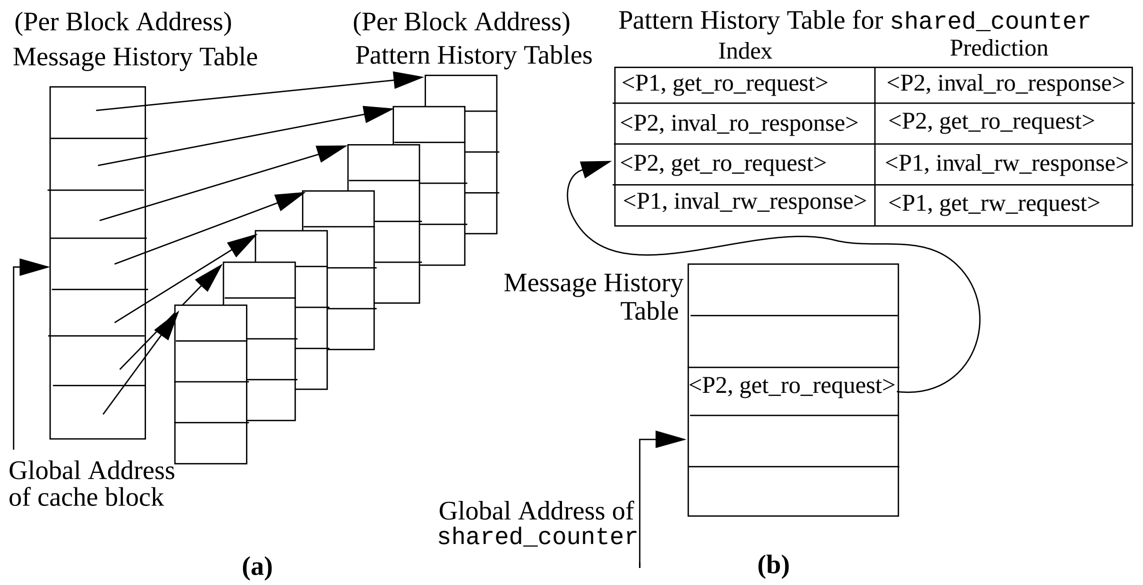


Figure 6-3. Cosmos' structure. (a) shows the logical structure the Cosmos coherence message predictor and (b) shows an example of how the message and pattern history tables for a directory may look like for the `shared_counter` in Figure 6-2. In this example, I assume that the last message received by the directory is a get_ro_request from the consumer (denoted as P2). So, Cosmos will predict the next message to be an inval_rw_response from the producer (denoted as P1).

Fortunately, a modified version of Yeh and Patt's two-level adaptive branch predictor called *PAp* [139] satisfies the above requirements! I call such a coherence message predictor *Cosmos*. Given the address of a cache block and the history of messages received for that block, Cosmos can predict with high accuracy the sender and type of the next incoming message for the same block. I allocate a Cosmos predictor for every cache or directory in the machine.

Figure 6-3a shows the logical structure of Cosmos. Cosmos is a two-level adaptive predictor. The first-level table—called the *Message History Table (MHT)*—consists of a series of *Message History Registers (MHRs)*. Each MHR corresponds to a different cache block address. An MHR contains a sequence of `<sender, type>` tuples corresponding to the last few coherence messages that arrived at the node for the specific cache block. I call the number of tuples maintained in each MHR the *depth* of the MHR.

The second-level table of Cosmos consists of a sequence of *Pattern History Tables (PHT)*, one for each MHR. Each PHT contains prediction tuples corresponding to possible MHR entries. Each PHT is indexed by the entry in the MHR entry. The next two subsections outline how to obtain predictions from and update entries in Cosmos.

Figure 6-3b shows the entries in an MHR and its PHT corresponding to the `shared_counter` variable in Figure 6-2. The MHT in Figure 6-3b has a depth of one, so this MHR entry contains only one `<sender, type>` tuple. The `<P2, get_ro_request>` tuple shown in this figure denotes that the last message received for the cache block containing the `shared_counter` is a `get_ro_request` message from the processor P2, which is consumer of the `shared_counter` in this case. The corresponding PHT captures patterns of messages received for `shared_counter`. For example, earlier Cosmos observed a `get_ro_request` message from processor P1 followed by an `inval_ro_response` from processor P2. The first entry of the PHT reflects this relationship. Thus, Cosmos will predict the arrival of an `inval_ro_response` message from processor P2, next time it sees a message `get_ro_request` from processor P1. Because the MHR contains the tuple corresponding to the last message received, to obtain a prediction I simply find the correct MHR, and use that entry to index into the PHT, which will give us a prediction if an entry exists for that tuple.

Cosmos borrows its two-level structure from Yeh and Patt's two-level adaptive branch predictor called PAp (see Section 6.1.2).[1] Nevertheless, Cosmos differs from PAp in three ways. First, the first-level table in Cosmos is indexed by the address of a cache block, whereas PAp is indexed by the program counter of a branch. Second, Cosmos must choose one prediction from several alternatives, whereas PAp usually chooses between two alternatives—branch taken or branch not taken. Third, the state machine in each PHT entry in PAp encodes the history of the last few outcomes of the same branch. Instead, a PHT entry

1. Wang and Franklin's data value predictor [129] uses a similar two-level structure. Unlike Cosmos, their first level table is indexed by the instruction address. Like Cosmos their second-level table is indexed by patterns from the first-level table.

in Cosmos simply consists of a prediction. Additionally, PHT entries in Cosmos can contain state machines (Section 6.2.6), but these are typically used as filters to remove noise from the incoming message stream.

Below I outline the exact steps involved in obtaining a prediction from and updating Cosmos.

6.2.3 Obtaining Predictions from Cosmos

Here the steps involved to obtain a prediction from Cosmos:

- index into the MHR table with address of a cache block,
- use the entry in MHR to index into the corresponding PHT, and
- return the prediction entry (if one exists) in the PHT as the predicted tuple, which contains the predicted sender and type of the next incomng message corresponding to that cache block; otherwise, return no prediction.

6.2.4 Updating Cosmos

Typically, I expect Cosmos to be updated after every message reception when I know for sure the `<sender, type>` tuple of a message. Here are the steps involved in updating Cosmos:

- index into the MHR table with the address of a cache block,
- use the entry in MHR to index into the corresponding PHT,
- write new `<sender,type>` tuple as new prediction for the index corresponding to the MHR entry, and
- left shift the `<sender,type>` tuple into the MHR for the cache block.

6.2.5 How Cosmos Adapts to Complex Coherence Message Streams?

Cosmos can adapt to complex message streams, such as the one outlined at the end of Section 6.2.1. If two `get_ro_request` messages arrive out of order from two different consumers (P1 and P2), the PHT table will contain the following two entries:

Index	**Prediction**
`<P1, get_ro_request>`	`<P2, get_ro_request>`
`<P2, get_ro_request>`	`<P1, get_ro_request>`

Therefore, Cosmos can effectively predict the next incoming coherence message, even though incoming messages may arrive in a different order in different instances. [1]

For more complicated sequences of incoming messages, Cosmos may need an MHR with depth greater than one. For example, if three `get_ro_request` messages come out of order from three consumers (P1, P2, and P3), then the PHT for a Cosmos predictor with MHR of depth = 2 may contain the following three entries:

Index	**Prediction**
`<P1, get_ro_request>, <P2, get_ro_request>`	`<P3, get_ro_request>`
`<P2, get_ro_request>, <P3, get_ro_request>`	`<P1, get_ro_request>`
`<P3, get_ro_request>, <P1, get_ro_request>`	`<P2, get_ro_request>`

Clearly, this allows Cosmos to predict the third incoming coherence message accurately based on the history of previous messages. Fortunately, several studies (e.g. [131, 136, 86]) have shown that the average number of sharers of a cache block is usually less than two. Consequently, I do not expect the depth of the MHR to be very high for most applications. Specifically, I found that an MHR of depth three is sufficient in most cases for the five parallel applications I study in this chapter.

6.2.6 Filtering Noise from Coherence Message Stream

When updating Cosmos we can use *filters* to reduce noise from the coherence message stream in the same way Yeh and Patt's PAp predictor removes noise from a stream of branches. For example, if 99% of the time, message B follows message A, then on seeing

1. A more aggressive predictor could ignore the senders for the `get_ro_request` messages. However, this may not be possible if there are intervening messages of other types for the same cache block.

message A, Cosmos will predict the next message to be B. I do not want the prediction to change if once in a while messages arrive in the sequence: A, C, and B, instead of the sequence A, B. Branch predictors have a similar problem when programs exit loops. Frequently, the exit from loops is a taken branch; however, when the loop is executed completely, the exit is a not-taken branch. Branch predictors typically avoid updating their prediction on exiting a loop via a two-bit saturating counter proposed by J. Smith [114]. One bit of the two-bit counter represents the direction of the branch and other bit represents the counter. Because a message needs more than one bit to represent a `<sender, type>` tuple, I simplify the counter and use only a single bit. With this single-bit counter, I update the prediction for a cache block to a different message only if we see two consecutive message mispredictions for the same block.

My results (Section 6.5.2) suggest that filters increase the prediction accuracy for Cosmos predictor with MHR depth of one, but they do not help Cosmos predictors with MHR depth greater than one. This is because both history and filters reduce noise from the message stream. However, history information adapts to the noise, while filters simply remove it.

6.2.7 Implementation Issues for Cosmos

Cosmos is a two-level adaptive predictor with the first level containing message history registers (MHRs) and the second level containing pattern history tables (PHT). It may be possible to merge the first-level table with the cache block state maintained at both directories and caches. However, this may lead to a loss of Cosmos' history information when cache blocks are replaced. This problem may not arise for the directory because directory state is usually persistent during the entire duration of a parallel application.

The second-level table is more challenging to implement because it may require large amounts of memory to capture pattern histories for each cache block. However, my results (Section 6.5.2) show that Cosmos' memory overhead for 128 byte cache blocks is less than 14% for an MHR depth of one. This is because the number of pattern histories corre-

Prediction	**Prediction Location**	**Static/ Dynamic**	**Action**	**Protocol**
Load/store from processor	Cache	Static	Prefetch block in shared or exclusive state	Stanford DASH protocol [69]
Read-modify-write	Directory	Static	Directory responds with block in exclusive state on read miss for an idle block	SGI Origin protocol [66]
Read-modify-write	Cache	Static	Cache requests exclusive copy on read miss	Dir_1SW [51], Dir_1SW+ [136]
Store from different processor	Cache	Static	Replace block and return to directory	Dir_1SW [51], Dir_1SW+ [136]
Store from different processor	Directory	Dynamic	Invalidate block and replace block to directory if exclusive	Dynamic Self-Invalidation [67]
Block migrates between different processors	Directory	Dynamic	On read miss return block to requesting processor in exclusive state	Migratory protocols [28, 120]

Table 6.2: Examples of prediction-action pairs in existing protocols.

sponding to a cache block is low, that is, less than four (on average) for an MHR depth of one for all five applications I studied in this chapter. Consequently, I could preallocate four pattern history entries corresponding to each cache block. If a cache block needs more pattern histories, then it can allocate them from a common pool of dynamically allocated memory in the same way LimitLESS [20] directory entries captures the list of sharers for a particular cache block. Nevertheless, higher prediction accuracies may require greater MHR depths, which may result in larger amounts of memory.

Clearly, storing, accessing, and updating these tables require moderate amounts of memory and computing power. However, the availability of tens of millions of on-chip transistors makes hardware implementations of these tables feasible. Additionally, software implementations of Cosmos is also practical because of the advent of symmetric multiprocessing (SMP) nodes in which the incremental cost of adding an extra processor for speculation is quite low (e.g. less than 5% of the cost of a node).

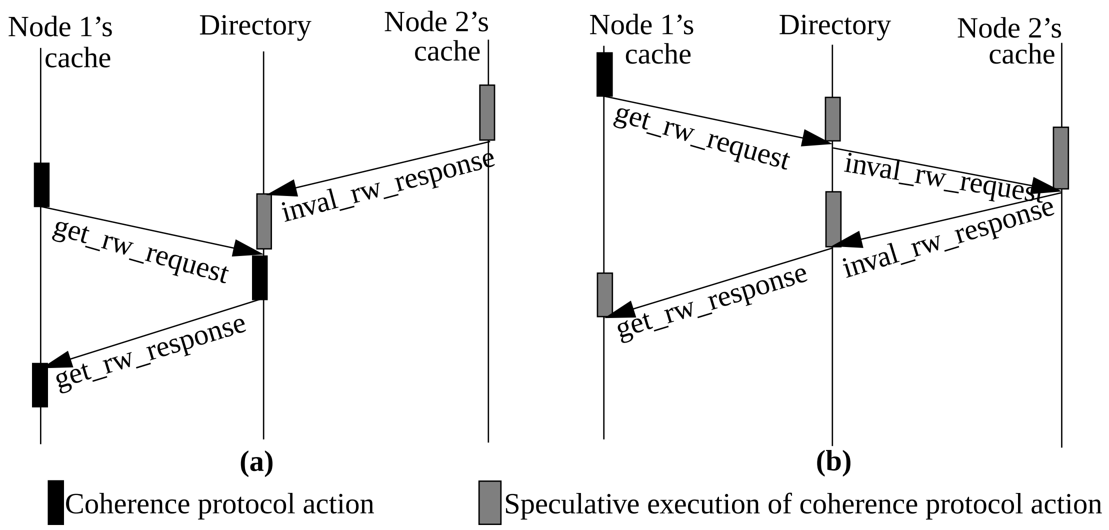


Figure 6-4. Two examples of using prediction to accelerate coherence protocols. (a) shows a protocol in which protocol actions are accelerated in anticipation of Node's 1 write miss. (b) shows a protocol that predicts incoming coherence messages, updates protocol state, generates (but does not send) messages speculatively, and commits protocol state and messages only if the predicted message arrives.

6.3 Using Coherence Protocol Message Predictors

This section briefly discusses how a coherence protocol message predictor, such as Cosmos, can be integrated with a coherence protocol. Predictors would sit beside each standard directory and cache module and accelerate coherence activity in two steps. First, they would monitor message activity and make a *prediction.* Second, based on the prediction, they will invoke an *action* in the standard coherence protocol. The key challenges include mapping predictions to actions (Section 6.3.1), performing actions at the right time (not too early or late) (Section 6.3.2), and dealing with mis-predictions (Section 6.3.3).

6.3.1 Mapping predictions to actions

Mapping predictions to actions is straightforward in many cases. Table 6.2 lists several examples of prediction-action pairs. For example, a directory action corresponding to a read-modify-write prediction for a block would be to return the block to the requesting cache in "exclusive" state, instead of the "shared" state.[1] Figure 6-4a shows another example where the predictor in node 2's cache predicts a write miss from another processor. A consequent action—as done by an implementation of Lebeck and Wood's dynamic self-invalidation protocol [67]—would be to replace the block from node 2's cache to the directory before the directory receives the write miss request from node 1's cache.

More generally, each directory and cache can predict incoming coherence messages, execute protocol actions speculatively (which may include sending messages speculatively), and take appropriate actions on mis-predictions (Figure 6-4b). Speculative execution of coherence protocol action may also involve executing a sequence of protocol actions, instead of executing a single action (that is normally done). This allows a directory and a cache to optimize for sharing patterns not known *a priori*. For example, a directory optimizing for a sequence of read-modify-write operations from different processors can directly capture the migratory protocol optimizations.

6.3.2 Detecting when to perform actions

Detecting when to perform actions is simple in some cases, but can be tricky in others. An obvious time to trigger actions would be to do so on certain protocol transitions. For example, the directory can trigger the action corresponding to a read-modify-write prediction when a read miss request arrives for a block. In Figure 6-4a, node 2's cache can trigger the block replacement action when it sees `inval_rw_request` messages for other spatially contiguous blocks. Alternatively, Cosmos predictions can be enhanced with a program counter that will give directories and caches to a more precise estimate of when to trigger actions.

6.3.3 Handling mis-predictions

Mis-predictions can leave the processor state, protocol state, or both in an inconsistent state. Consequently, a protocol must recover from mis-predictions. In general, actions can

1.Cosmos identifies a read-modify-write operation from the signature: `<P,get_ro_request>`, `<P,upgrade_request>`.

be classified into three categories. Below I outline possible recovery mechanism for each action.

- Actions that move protocol between two “legal” states require no recovery on mis-prediction. Replacement of a cache block that moves the block from “exclusive” to “invalid” state is an example of such an action (Figure 6-4a).
- Actions that move the protocol state to a “future” state, but do not expose this state to the processor can recover from mis-predictions transparently. This scheme is analogous to the “future file” scheme to implement precise exceptions [115]. On detecting a mis-prediction a protocol simply discards the future state. On detecting a prediction success, however, the coherence protocol state must commit the future state and expose it to the processor. Mis-predictions corresponding to actions in Figure 6-4b can use such recovery actions.
- Actions that allow both processor and protocol states to move to future states need greater support for recovering from mis-predictions. One possible scheme is analogous to the “history file” scheme used to implement precise exceptions [115]. Before speculation begins both the processor and the protocol capture their states in a history file. Then, on detecting a mis-prediction both processor and coherence protocol must roll back to the state captured in the history file. On detecting a success, the current protocol and processor states must be committed. Such actions can be created by coupling a speculative processor, such as the MIPS R10000 [83], with a coherence protocol accelerated with prediction. This is perhaps the most aggressive form of speculation with a coherence protocol.

Directories and caches can detect prediction success or failure—as required in the last two actions—by simply verifying whether the next message for a cache block is indeed the predicted message or not. Additionally, if any of the last two actions generate messages that are sent speculatively to other directories or caches, then these directories or caches must be informed of the mis-prediction. This allows a directory or a cache to recover from mis-predictions caused by other directories and caches.

Benchmarks	**Iterations**
appbt	30
barnes	19
dsmc	320
moldyn	40
unstructured	10

Table 6.3: Number of iterations for each benchmark. .

6.4 Methodology

I evaluate Cosmos' prediction accuracy using traces of coherence messages obtained from the Wisconsin Stache protocol (Section 6.4.1) running five parallel scientific applications—*appbt*, *barnes*, *dsmc*, *moldyn*, and *unstructured* (Section 4.2.3).[1] Each application has a start-up phase to initiate the computation (e.g. initiate data structures). My traces do not contain coherence messages generated in this start-up phase. Also, the applications in this chapter use a number of iterations (see Table 6.3) different from those listed in Table 4.3 to allow Cosmos to adapt to sharing patterns of these applications (Section 6.5.2).

I generated the traces from the Wisconsin Wind Tunnel II simulator [92] simulating a 16-node parallel machine, with each node having one processor, a coherent memory bus, and a $CNI_{32}Q_m$ network interface [85]. The system parameters used to collect these traces are the same as in Chapter 4 and Chapter 5 (see Table 4.2). Cosmos' prediction accuracy, however, is largely insensitive to variations in system parameters, such as network latency. For example, changing the network latency from 40 nanoseconds to one microsecond hardly changes Cosmos' prediction rates for the five applications I study in this chapter.

1. Of the seven applications used in Chapter 4 and Chapter 5, I excluded *em3d* because it would not run with 128-byte Stache blocks (Chapter 6.4.1) and *spsolve* because I did not have access to a transparent shared-memory version of this program.

6.4.1 Wisconsin Stache Protocol

I obtained my coherence message traces from the Wisconsin Stache protocol [100]. Stache is a software, full-map, and write-invalidate directory protocol that uses part of local memory as a cache for remote data. Currently, Stache is implemented on the Tempest interface [52], which is a portable interface for writing shared-memory programs. Table 6.1 shows all the types of coherence messages generated by Stache. These coherence messages are also common to most full-map, write-invalidate directory protocols. For all my simulations with Stache I use a (software) cache block size of 128 bytes.

Stache differs other full-map, write-invalidate coherence protocols in five ways:

- Unlike the DASH protocol, Stache uses the *half-migratory optimization*. In this optimization a directory requests a cache to mark an exclusive block invalid, and not shared, when it receives a read or write miss request from another cache. This is beneficial if this same cache block is not immediately read from the former cache.
- The Stache implementation I use in this thesis allocates pages in round-robin fashion across the 16 nodes. The owner of each page functions as the directory for that page. The directory pages are optimized to function as cache pages for the local node. Consequently, in most cases Stache does not generate local messages between the cache and directory within a particular node.
- Cache blocks on a cache page in a local node communicate only with one specific directory page in another node. Consequently, for blocks on a cache page, the sender is always a fixed node containing the directory page. A directory page can, however, receive messages from any node caching the page.
- Currently, Stache does not replace pages (and, hence, cache blocks) from the portion of local memory it designates as a cache for remote memory. This implies that Cosmos' history information for cache blocks persists over time. Protocols that replace cache blocks may need to preserve the history information even after the block is

Depth of MHR	*appbt*			*barnes*			*dsmc*			*moldyn*			*unstructured*		
	C	**D**	**O**	**C**	**D**	**O**	**C**	**D**	**O**	**C**	**D**	**O**	**C**	**D**	**O**
1	91	77	**84**	80	42	**62**	94	73	**84**	92	79	**86**	85	65	**74**
2	90	79	**85**	81	56	**69**	95	77	**86**	91	80	**86**	90	86	**88**
3	89	80	**85**	79	57	**69**	94	92	**93**	90	79	**85**	90	88	**89**
4	89	80	**85**	78	56	**68**	94	92	**93**	90	77	**84**	96	88	**92**

Table 6.4: Cosmos' prediction rates (expressed in percentage of hits). Depth of MHR denotes the number of messages used by Cosmos to predict the next incoming coherence message. C = prediction rate at cache, D = prediction rate at directory, and O = overall prediction rate.

replaced. Alternatively, such protocols can speculate only at the directory, where Cosmos' history information is persistent during the duration of a parallel application.

- Barriers are implemented with point-to-point messages. Consequently, Cosmos' prediction accuracies do not include prediction rates for barrier variables.

Nevertheless, I see no reason why Cosmos prediction results with Stache should be significantly different from what would be obtained with a full-map, write-invalidate directory protocol.

6.5 Results

In this section I examine Cosmos' basic prediction accuracy (Section 6.5.1) and then delve into Cosmos' sensitivity to noise and initialization effects and Cosmos' memory requirements (Section 6.5.2).

6.5.1 Basic Prediction Rate

Table 6.4 shows that Cosmos achieves high prediction accuracy. With an MHR depth of one, Cosmos' overall prediction accuracy ranges between 62-86%. Cosmos achieves such high accuracy because cache blocks in most applications generate predictable coherence message signatures. These signatures are related directly to sharing patterns of an application's data structures. All the applications, except *barnes*, have one or more fixed signatures (see Figures 6-5 and 6-6) throughout the entire execution of the parallel application. *Barnes* has slightly lower accuracy because shared-memory addresses are reassigned to different objects across iterations. Below I discuss each application's prediction accuracy in detail.

Table 6.4 shows that Cosmos has higher accuracy for a cache compared to a directory. For the Stache protocol, a cache receives messages from a fixed sender—that is, a fixed directory, which limits the number of `<sender,message-type>` tuples Cosmos must choose its predictions from. In contrast, a directory receives messages from multiple caches (i.e. senders) for the same cache block. Consequently, Cosmos' predictions are more accurate for Stache caches than Stache directories.

Table 6.4 also shows that Cosmos' prediction accuracy usually increases with the increase in the MHR depth. With MHR depth of two, the accuracy ranges between 69-88%, while a depth of three results in prediction accuracy that ranges between 69-93%. Having history information helps because it allows Cosmos to recognize predictable coherence streams (Section 6.2.5). However, most of the applications do not benefit beyond an MHR depth of three (Table 6.4).

Below I examine why Cosmos achieves high prediction rates for each of the five applications. Surprisingly, variations in simple sharing patterns studied by Bennett, et al. [9] and Weber and Gupta[46], can lead to sequences of coherence actions (and consequent signatures) that are significantly different from those generated by simple sharing patterns (e.g. see *unstructured*'s sequence of messages below). Consequently, predictors based on simple sharing patterns may not be able to correctly speculate the sequence of coherence actions that may be generated. However, Cosmos can capture such variations in sharing patterns because Cosmos adapts to the incoming message stream, which directly determines the sequence of coherence actions to follow.

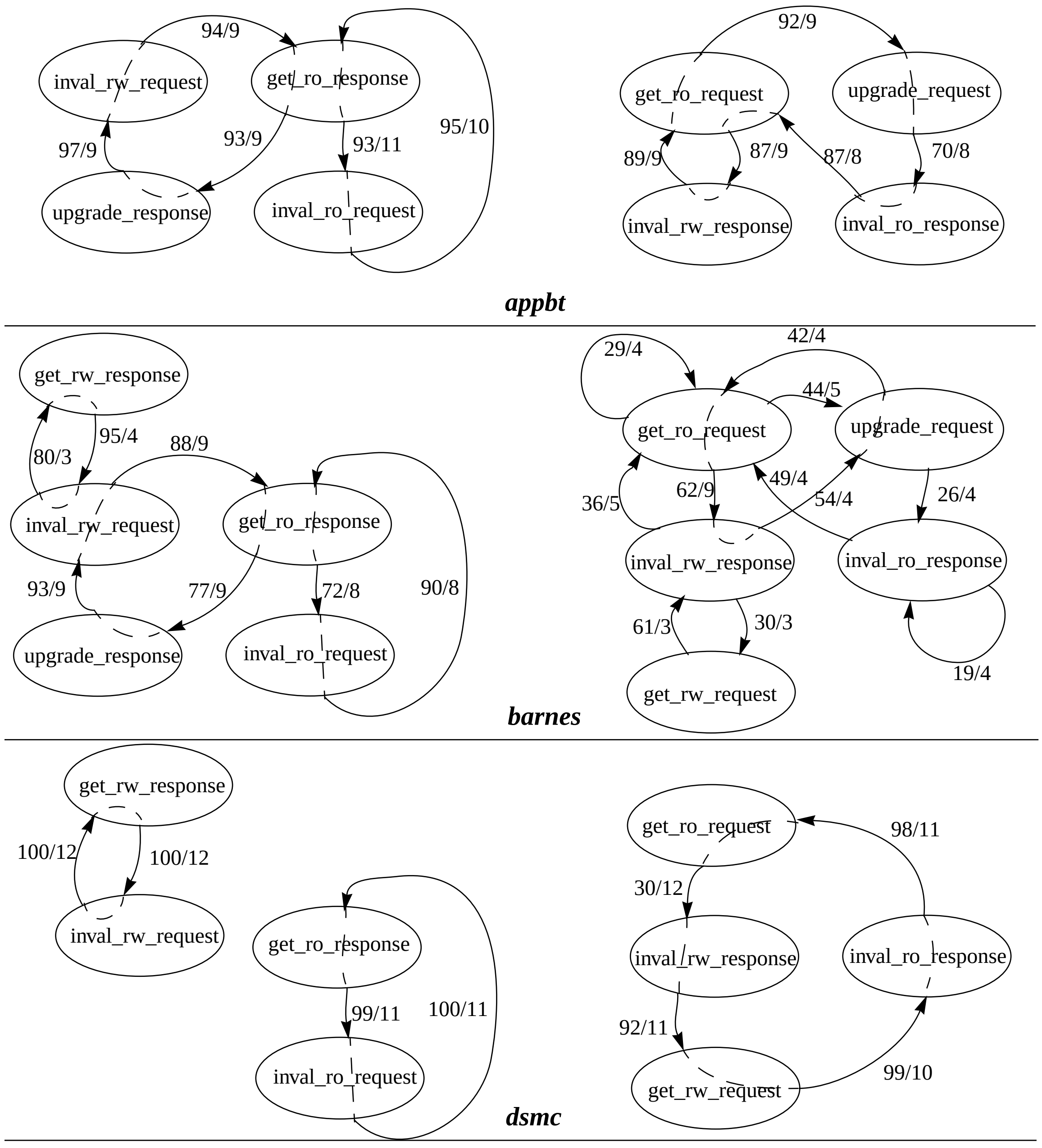


Figure 6-5. Dominant (incoming) message signatures for *appbt*, *barnes*, and *dsmc*. Arcs represent the order in which two messages arrived. Each arc is labelled as X/Y, where X = percentage of correct predictions for that particular arc and Y = percentage of references to that arc. All X and Y numbers are measured with a Cosmos predictor with MHR depth of one. The left side shows the transitions for the cache and right side shows transitions for the directory. All Y for a benchmark do not add up to 100% because I only present the dominant transitions I observe. The dotted lines represent dominant message signatures observed in the message stream.

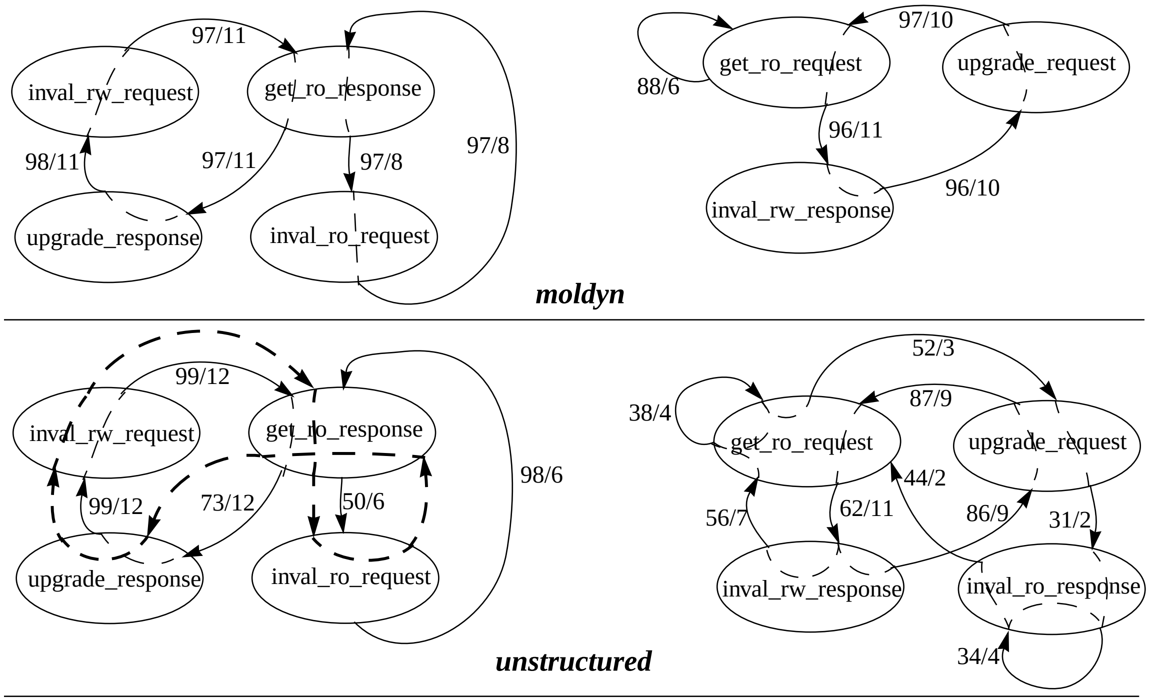


Figure 6-6. Dominant (incoming) message signatures for *moldyn* and *unstructured*. See caption of Figure 6-5 for an explanation of the figure. I show *unstructured*'s second dominant message signature (at the cache) using bold and dashed lines.

***Appbt*'s** high prediction accuracy results from its producer-consumer sharing pattern. *Appbt* is a three-dimensional stencil-style code in which a cube is divided up into subcubes. Each subcube is assigned to one processor. Communication occurs between neighboring processors along boundaries of the subcubes.

The sharing pattern that results in the sequence of messages shown for *appbt* in Figure 6-5 is: producer reads, producer writes, and consumer reads. This pattern repeats for most cache blocks throughout the entire application. Consequently, Cosmos adapts well to *appbt* resulting in a prediction accuracy of 85%. Note that the half-migratory optimization discussed in Section 6.4 hurts here because the producer first reads a block before writing to it. In the absence of this optimization, the producer pattern would have simply cycled through the two messages: `inval_rw_request` and `upgrade_request`.

Figure 6-5 shows that all transitions for *appbt* have high prediction accuracy except the transition from `upgrade_request` to `inval_ro_response` at the directory. The low accuracy on this transition results from false sharing in two data structures. It appears that this false sharing generates multiple signatures that the protocol oscillates between randomly. This confuses the predictor resulting in lower accuracy. Perhaps a “fuzzy” predictor that predicts multiple signatures (with different probabilities) simultaneously can track this false sharing better.

***Barnes*'s** prediction accuracy ranges between 62-69% for different MHR depths. This is slightly lower than that for the other applications. I suspect this happens in *barnes* because nodes of the octree in *barnes* are reassigned to different shared-memory addresses in different iterations. Unfortunately, Cosmos cannot make accurate predictions for the nodes of the octree because its prediction is based on information it collected on past behavior (e.g. previous iterations) of a particular shared-memory address (at a cache block granularity).

I suspect that *barnes*’ low prediction accuracy results from such reassignment because of three reasons. First, bodies, which are not reassigned, exhibit significantly higher prediction accuracy than the nodes. Second, increasing the depth of MHR (Table 6.4) or the number of iterations (see “Time to Adapt” in Section 6.5.2) does not increase the prediction accuracy of *barnes*. If *barnes*’ low prediction accuracy resulted from other reasons, such as random traversal of the octree, then increasing the MHR depth or number of iterations would steadily increase Cosmos’ prediction accuracy because Cosmos adapts to new (and even random) patterns. Finally, Figure 6-7 shows that the distribution of number of bodies in *barnes* in different children of the root varies widely across different iterations. This large variation suggests a large degree of reassignment in the octree, which can potentially lead to the lower prediction accuracy for the nodes of the octree.

Figure 6-5 shows that *barnes* has a variety of sharing patterns, some of which exhibit dominant signatures throughout the execution of the program. However, the low accuracies on most arcs improve with more history information (i.e. greater MHR depth).

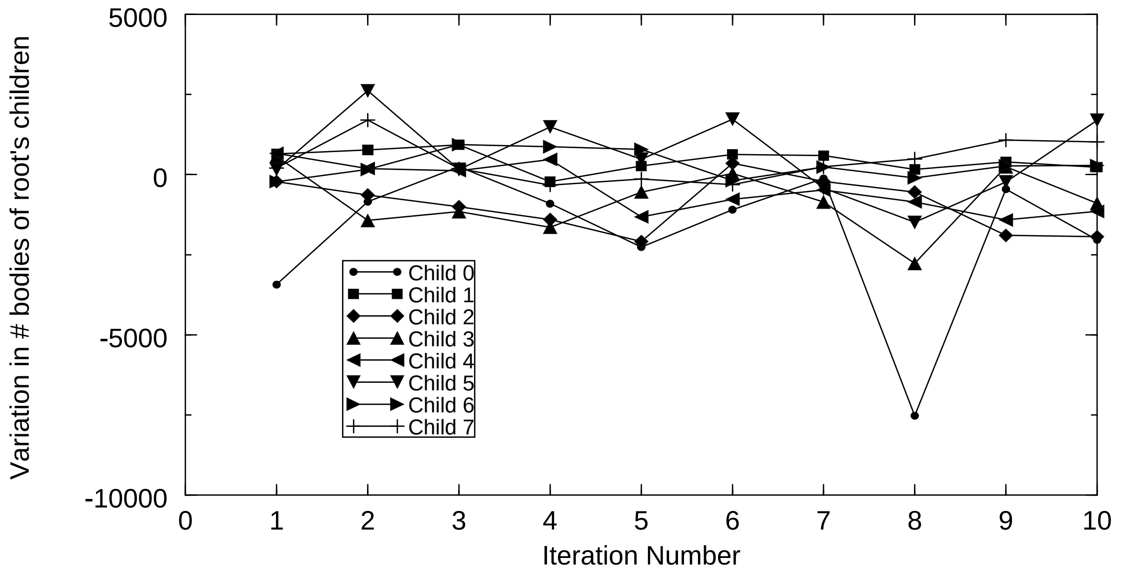


Figure 6-7. Variation in number of bodies in the root's children in *barnes*. The root has eight children because this is an octree.

Dsmc shows the highest accuracy among all the five applications. *Dsmc*'s dominant sharing pattern is the classical producer-consumer pattern in which the producer writes and the consumer reads shared cache blocks. This happens because at the end of each iteration *dsmc* communicates information between two processors via shared buffers. This leads to the message sequence shown in Figure 6-5. Note that the half-migratory optimization helps *dsmc* because the producer does not read the data before it writes to it. Consequently, invalidating the producer's cache blocks, instead of converting them to read-only, avoids an extra handshake with the directory.

Figure 6-5 shows that the transition from `get_ro_request` to `inval_rw_response` has a low prediction accuracy. However, this low accuracy disappears with increased MHR depth. This happens because updates to shared buffers frequently follow deterministic patterns; but, in some cases a processor must lock a shared buffer before writing to it. This creates somewhat oscillating patterns that confuses Cosmos. Fortunately, Cosmos learns to isolate these cases using either more history information or via noise filters (see Section 6.5.2).

D	*appbt*			*barnes*			*dsmc*			*moldyn*			*unstructured*		
	0	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**
1	84	85	85	62	66	66	84	86	86	86	86	86	74	78	78
2	85	85	86	69	71	71	86	88	88	86	86	86	88	89	89

Table 6.5: Cosmos' prediction accuracy with filters. This table shows the prediction accuracy of Cosmos as I vary the maximum count of the saturating counter from 0 to 2. The saturating counter filters noise from the coherence message stream (Section 6.2.6). The overall prediction rates in Table 6.4 correspond to this table's column 0 (i.e. no filter).

***Moldyn*'s** high accuracy results from two dominant sharing patterns: migratory and producer-consumer patterns. The migratory sharing pattern results in the message sequence `<get_ro_response, upgrade_response, inval_rw_response>` in both processors a block is migrating between. The same pattern is exhibited for the producer in the producer-consumer pattern. However, the consumer for the producer-consumer pattern sees the sequence: `<get_ro_response, inval_ro_request>`. Hence, the number of references to the pattern `<get_ro_response, upgrade_response, inval_rw_response>` is greater than the number of references to the pattern `<get_ro_response, inval_ro_request>` (Figure 6-6). The sequence seen at the directory results primarily from the migratory pattern.

Moldyn's migratory pattern results from the way it reduces a shared array, which contains force calculations for simulated molecules. In each iteration each processor collects its contribution for different elements of the shared array in a private array. At the end of the iteration each processor adds its contribution from the private array to the shared array. Updates to each element in the shared array happens in a critical section, which results in the migratory pattern.

Moldyn's producer-consumer sharing pattern results from updates to a shared array that contains the coordinates of simulated molecules. *Moldyn*'s producer-consumer pattern results in message signatures similar to that of *appbt*'s at both the producer and consumer caches. However, the overall number of consumers for *moldyn* is 4.9, whereas for *appbt*

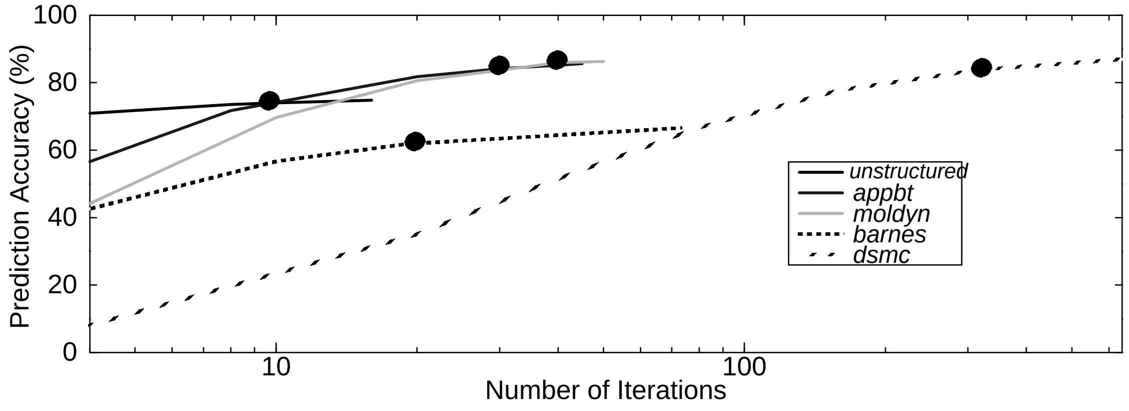


Figure 6-8. Cosmos' time to adapt. This figure shows how many iterations it takes Cosmos (with MHR depth of one and no filter) to reach steady-state prediction accuracy. For each application the black dot indicates the number of iterations I have chosen for all my other experiments in this chapter. The time to achieve steady-state behavior for other Cosmos predictors (with different MHR depths and filters) is similar the Cosmos predictor shown in this figure.

the number of consumers is one. Consequently, directory observes back-to-back `get_ro_request` messages arriving with high predictability.

Unstructured is different from the rest of the applications because it has different dominant signatures for the same data structures in different phases of the application. The same data structures oscillate between migratory and producer-consumer sharing patterns. The migratory sharing pattern is similar to *moldyn*'s and occurs when each processor updates different elements of the shared arrays in critical sections. The migratory pattern is followed by the producer-consumer pattern in which a producer is itself a consumer of the data. The average number of consumers per producer is 2.6. The signature shown in bold and dashed arrows in Figure 6-6 represents the transition from migratory to producer-consumer pattern. The directory sees corresponding signatures.

Figure 6-6 shows that *unstructured*'s prediction accuracy for several arcs with MHR depth of one is low. This is because of the change in sharing pattern. Table 6.4 shows, however, that Cosmos' accuracy increases from 74% to 92% as the MHR depth increases

Transition	4 iterations		80 iterations		320 iterations	
	hits	refs	hits	refs	hits	refs
`<get_ro_response, upgrade_response>`	2%	20%	34%	4%	62%	2%
`<get_ro_request, inval_rw_response>`	2%	25%	18%	13%	30%	12%
`<inval_rw_response, upgrade_request>`	1%	19%	18%	4%	35%	1%

Table 6.6: *Dsmc*'s prediction accuracies for specific transitions. This table shows *dsmc*'s prediction accuracy for different number of iterations. refs is percentage of total references to the transition. hits is the percentage of hits to the transition. These numbers are measured with a filterless Cosmos predictor with MHR depth of one.

from one to four. This increase in prediction accuracy from the increase in MHR depth also results in high prediction accuracies for these arcs.

6.5.2 Additional Analysis

Effect of Filters on Prediction Accuracy. Noise filters can increase the prediction accuracy of Cosmos. I implement Cosmos' noise filter as a saturating counter, which counts upwards from zero and saturates at a maximum count. Table 6.5 shows the prediction accuracy of Cosmos as I vary the maximum count between 0 and 2.

Filters increase prediction accuracy slightly (up to around 6%) only for Cosmos predictors with MHR depth of one. For MHR depth of two or beyond filters do not help much. This is because both filters and history information remove noise from the message stream. However, history information allows Cosmos to learn from and adapt to the noise. Consequently, if the noise repeats, then Cosmos can achieve higher accuracy. In contrast, filters simply remove noise, but do not let Cosmos adapt to it. Hence, predictors with filters and MHR depth of one achieve lower accuracy than predictors with greater MHR depths. Additionally, filters do not help predictors that have MHR depth greater than one.

Time to Adapt. A critical question for predictors, such as Cosmos, is how long it takes them to achieve the steady-state prediction rates. Cosmos predictors need time to achieve steady-state behavior because they adapt to the incoming stream. I use number of itera-

Depth of MHR	*appbt*		*barnes*		*dsmc*		*moldyn*		*unstructured*	
	Ratio	**Ovhd**	**Ratio**	**Ovhd**	**Ratio**	**Ovhd**	**Ratio**	**Ovhd**	**Ratio**	**Ovhd**
1	1.2	5.4%	3.8	13.5%	0.8	3.9%	0.8	4.0%	1.7	6.8%
2	1.4	9.6%	6.9	35.4%	0.4	5.1%	1.1	8.3%	2.1	12.8%
3	1.9	16.4%	9.3	63.0%	0.3	6.7%	1.6	14.9%	2.8	21.9%
4	2.6	26.5%	10.9	91.8%	0.3	8.9%	2.0	21.6%	3.4	33.0%

Table 6.7: Memory overhead of Cosmos predictors (with no filter). Ratio = total number of PHT entries / total number of MHR entries. MHR entries correspond to cache blocks that were referenced at least once in the parallel section of an application. Ovhd expresses the average memory overhead per 128-byte block as a percentage of the block size. More precisely, Ovhd = (tuple size * [MHR depth + Ratio * (MHR depth + 1)] * 100 / 128)%. I assume the tuple size of two bytes (12 bits for processors and 4 bits for coherence message types). Note that some Ratios are less than one. This is because unless the number of protocol references to a cache block is greater than the MHR depth, I do not allocate a PHT for that MHR. This makes all of *dsmc*'s Ratios less than one because some of *dsmc*'s shared-memory data structures are touched rarely. For the same reason, unlike other benchmarks, *dsmc*'s Ratio decrease with increase in MHR depth because the number of these shared-memory blocks that are touched more times than the MHR depth is even fewer.

tions of each application as an approximation to time. This is because the five parallel applications I examined in this chapter iterate over a number of steps or iterations. Cosmos can predict incoming coherence messages for a cache block fairly accurately because sharing pattern of a cache block in one iteration is usually similar to its sharing pattern in the previous iteration.

Figure 6-8 shows that *unstructured* and *barnes* achieve steady-state behavior quickly (in less than 20 iterations. *Appbt* and *moldyn* take slightly longer (around 30 iterations). *Dsmc*, however, takes a large number of iterations (around 300) to achieve steady state prediction rates. This is because specific transitions in *dsmc* take a large number of iterations to achieve reasonable prediction accuracies (Table 6.6).

Memory Requirement of Cosmos Predictors. Table 6.6 shows that dynamic memory overhead incurred by Cosmos predictors is acceptable—that is, less than 22%—for most applications for predictors with MHR depths of three or lower. Additionally, the number

Depth of MHR	**appbt**		***barnes***		***dsmc***		***moldyn***		***unstructured***	
	Base	**+P**	**Base**	**+P**	**Base**	**+P**	**Base**	**+P**	**Base**	**+P**
1	84	-13	62	+5	84	+8	86	+1	74	-1
2	85	+2	69	+8	86	+7	86	+1	88	-3
3	85	+3	69	+7	93	+1	85	+3	89	-2
4	85	+2	68	+10	93	+1	84	+5	92	+0

Table 6.8: Using processor numbers to improve Cosmos' accuracy. Depth of MHR denotes the number of messages used by Cosmos to predict the next incoming coherence message. Base = base overall prediction rates that appear in Table 6.4. +P denotes the increase in the prediction rate when we combine processor (sender) ids with cache block addresses in the MHRs (Section 6.6).

of PHT entries per cache block (or MHR entry) is less than three in most cases. The low PHT to MHR ratio suggests that perhaps a scheme that statically allocates three or four PHT entries per cache block and dynamically allocates the rest from a common pool of memory may work. Only for *barnes* the memory overhead is as high as 63% for MHR depth of three because *barnes* reassigns shared-memory addresses to logically different objects, which confuses Cosmos and leads to greater number of coherence message patterns

6.6 Increasing Cosmos' Accuracy

Cosmos' accuracy can be increased further by combining sender processor numbers with cache block addresses in the MHR (first-level table). The intuition behind this is that not only incoming coherence messages *for a specific cache block* are highly correlated (done so far), but also incoming coherence messages *from a specific processor for a specific cache block* are highly correlated. [1]

In the base model discussed so far, given a cache block address (index for MHRs) and the history of messages (i.e. <sender processor, message type> tuples), Cosmos can predict with high accuracy the <sender processor, message type> tuple of the next message

1. Guri Sohi suggested this approach. This improvement does not appear in [90].

destined for the cache block. Cosmos' accuracy increases further if the MHRs are indexed by both cache block addresses and sender processor numbers. Thus, MHRs will be indexed by <cache block address, sender processor>, instead of <cache block address>, and the PHTs will predict <message type>, instead of <sender processor, message-type>.

Table 6.8 shows using sender processor numbers along with cache block addresses provides modest improvement in prediction accuracy, particularly for *barnes* and *dsmc*. The improvement arises almost entirely from the improvement in prediction accuracy at the directory.

6.7 Comparison with Directed Optimizations

In this section I compare Cosmos with directed optimizations—that is, optimizations introduced in a coherence protocol for specific sharing patterns. Dynamic self-invalidation [67] and migratory protocols [28, 120] are examples of two such protocols. Both can be thought of as implementing predictors directed at specific optimizations. Cosmos could be less cost-effective than predictors for directed optimizations because Cosmos requires more hardware resources to store, access, and update the Message History and the Pattern History Tables. However, it may be possible to reduce Cosmos' memory requirements by grouping predictions for multiple cache blocks together (similar to Johnson and Hwu's *macroblocks* [57]).

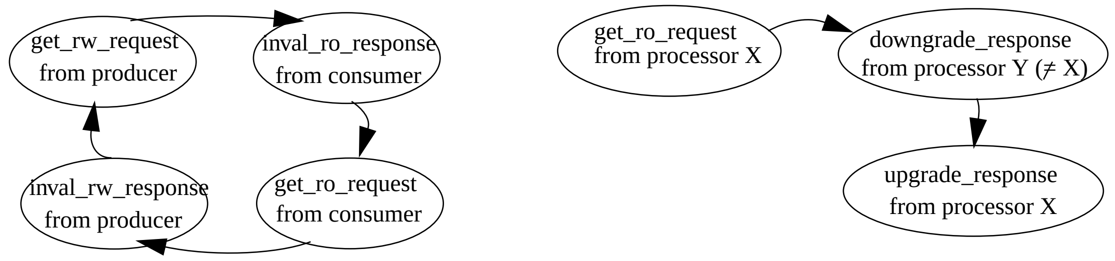
Cosmos could be better than directed optimizations for two reasons. First, including the composition of predictors of several directed optimizations in a single protocol could be more complex than Cosmos. All the predictors in existing coherence protocols that I am aware of are integrated with the finite-state machine of the coherence protocol. Such integration works well when one considers these protocols individually. Unfortunately, combining multiple such predictors into a single protocol can lead to an explosive number of interactions and states, which can make the resulting protocol bulky and hard to debug [21]. More critically, extending a bulky protocol with other kinds of speculation becomes even harder. In contrast, Cosmos captures the predictors for directed optimizations in a single predictor. Figure 6-9 shows the coherence message signatures that trigger the dynamic self-invalidation and migratory protocols. Cosmos can capture these signatures easily. Additionally, protocols accelerated with Cosmos are easier to extend because Cosmos separates the predictor from the protocol itself.

Second, Cosmos can discover application-specific patterns not know *a priori*. For example, Section 6.5.1 shows that one of *unstructured*'s signatures is a complex composition of migratory and producer-consumer sharing patterns. Predictors directed only at migratory or producer-consumer pattern will fail to track *unstructured*'s transition between migratory and producer-consumer sharing patterns. As Section 6.5.1 also shows, Cosmos can easily capture, filter, and adapt to different message signatures generated by variations in simple sharing patterns studied by Bennett, et al. [9] and Gupta and Weber [46].

6.8 Summary and Conclusions

This chapter explores using prediction to accelerate coherence protocols. A coherence protocol can execute faster if it can predict future coherence protocol actions and execute them speculatively. It shares with branch prediction the need to have a sophisticated predictor. The first contribution of this chapter is the design of the *Cosmos* coherence message predictor. Cosmos predicts the next `<processor,message-type>` in two steps reminiscent of Yeh and Patt's two-level *PAp* branch predictor. Cosmos faces a greater challenge than branch predictors because the Cosmos' prediction is a multi-bit `<processor,message-type>` tuple rather than a single branch outcome bit.

The second contribution of this chapter is a detailed evaluation of the Cosmos coherence message predictor. Using five scientific benchmarks on a target shared-memory machine with 16 processors running the Stache directory protocol, variations of Cosmos predict the source and type of the next coherence message with surprisingly-high accuracies of 62-69% (*barnes*), 84-86% (*moldyn*), 84-85% (*appbt*), 74-92% (*unstructured*), and 84-93% (*dsmc*).


get_rw_request from producer
inval_ro_response from consumer
get_ro_request from consumer
inval_rw_response from producer
get_ro_request from processor X
downgrade_response from processor Y (≠ X)
upgrade_response from processor X

(a) A self-invalidation signature **(b)** A migratory protocol signature

Figure 6-9. Signatures for dynamic self-invalidation and migratory protocols. The downgrade_response, not shown in Table 6.1, is a response to a downgrade_request sent by the directory. On receiving a downgrade_request for a block, a cache must change the block from exclusive to shared state.

Cosmos' high prediction accuracy results from predictable coherence message patterns or *signatures* associated with specific cache block addresses. Such signatures are generated by sharing patterns that do not change or change very slowly during the execution of these applications. Cosmos is more general than directed optimizations, such as dynamic self-invalidation and migratory protocols. Cosmos could be less cost-effective than the directed optimizations because it uses more resources (e.g., tables). Cosmos could be better than directed optimizations because (1) including the composition of these optimizations could be more complex than Cosmos and (2) Cosmos can discover and track application-specific patterns not known *a priori*.

More work is needed to determine whether the high prediction rates of Cosmos can significantly reduce execution time with a coherence protocol. This is work is analogous to taking a branch predictor with high prediction rates and integrating it into a micro-architecture to see how much it affects the bottom line. I believe that results in this chapter on Cosmos's high prediction rates indicate that work on the next step is justified.

Chapter 7

Thesis Summary

A modern computer system often “communicates” with a communication network more than it “computes.” Consequently, today much of a computer’s value depends on how well it communicates with external networks. As processors and networks continue to improve rapidly, interactions between a processor and a network interface (NI) becomes a dominant component of the overall communication latency.

An NI is a device that allows a processor to send and receive messages from a network. Conventional NIs suffer from several sources of high latency because they were designed with an interface similar to a disk’s interface. For example, conventional NIs are usually accessed via low-level software (e.g. device driver) inside the operating system, located on slow I/O buses, and accessed via direct memory access (DMA) or uncached, memory-mapped device registers. With current technology each of these components can incur between ten and hundreds of microseconds of latency.

This thesis investigates novel techniques to improve processor-NI interactions in parallel computer connected via a System Area Network (Appendix A). A key principle underlies these techniques: *treat NI access as regular, side-effect-free memory access, and not as a disk interface access*.

This thesis makes four contributions. The first contribution of this thesis is to show that treating NI access like a regular memory access opens up at least eight opportunities for improving processor-NI interactions. These opportunities are:

- using virtual memory hardware, and not operating system intervention, to virtualize the NI,
- placing the NI on the higher performance memory bus, and not on the slower I/O bus,
- using virtual memory as a huge buffer for network messages, instead of small amounts of dedicated memory on the NI,
- caching messages in processor and NI caches, like regular cachable memory,
- allowing out-of-order accesses and speculative loads on a processor's accesses to an NI, like side-effect-free regular memory accesses,
- transferring messages between processor caches, NI cache, and main memory through cache block transfers, instead of DMA,
- designing the application programming interface (or API) to the NI as memory-based queues, and not directly exposing the underlying data movement primitives as the API, and
- notifying processor of NI events through cache invalidations, instead of heavy-weight interrupts.

The second contribution of this thesis is the design and detailed evaluation of a novel class of NIs called *Coherent Network Interfaces* (CNIs). CNIs are the embodiment of the fundamental principle enunciated in this thesis. CNIs appear to their hosts more like memory than like a disk interface and, hence, exploit all eight opportunities for improving processor-NI interactions. CNIs use two mechanisms, *cachable device registers* and *cachable queues*, which interact with the host via cachable, coherent memory operations. CNIs use several optimizations—*lazy pointer*, *shadow head*, *sense reverse*, *empty entry removal*, *intra-message prefetch*, *dead message elimination*, and *cache bypass*—to further optimize processor-CNI interactions.

I performed a detailed simulation of four CNIs with a more conventional NI—that is, a Thinking Machines' CM-5 NI [124]—using a 16-node parallel machine, two microbenchmarks, and seven parallel scientific applications. For small message sizes—between 8 and 256 bytes—CNIs improve the round-trip latency by 87-342% compared to a conventional NI on a coherent memory bus. For moderately large messages, between 8 and 4096 bytes, CNIs improved the bandwidth by 109-202%. Results with the seven applications show that CNIs can improve performance by up to 21-190% compared to a conventional NI.

The third contribution of this thesis is a systematic classification and evaluation of two of the eight opportunities—data transfer and buffering—that underlie high-performance NI designs. I evaluate these parameters in the context of seven NIs, which abstract the data transfer and buffering parameters of the NIs in the Thinking Machines' CM-5, Fujitsu AP3000, Princeton User-Level DMA, Digital Memory Channel, MIT StarT-JR, and two CNIs ($CNI_{512}Q$ and $CNI_{32}Q_m$).

My results show that a high-performance NI design should effectively use the block transfer mechanism of the memory bus, minimize processor involvement for data transfer, directly transfer messages between an NI and the processor (at least in the common case), provide plentiful buffering (possibly in main memory), and minimize processor involvement to buffer incoming network messages. $CNI_{32}Q_m$ performs the best among the seven NIs because it effectively optimizes the data transfer and buffering parameters.

The fourth contribution of this thesis is the design of the *Cosmos* coherence message predictor. Unlike the rest of the thesis, this part focuses on shared-memory multiprocessors. Most large shared-memory multiprocessors use directory protocols to keep per-processor caches coherent. Some memory references in such systems, however, suffer long latencies for misses to remotely-cached blocks. To ameliorate this latency, researchers have augmented standard coherence protocols with optimizations for specific sharing patterns, such as read-modify-write, producer-consumer, and migratory sharing. This paper seeks to replace these directed solutions with general prediction logic that monitors coherence activity and triggers appropriate coherence actions.

This thesis takes the first step toward using general prediction to accelerate coherence protocols by developing and evaluating the *Cosmos* coherence message predictor. Cosmos predicts the source and type of the next coherence message for a cache block using logic that is an extension of Yeh and Patt’s two-level *PAp* branch predictor. For five scientific applications running on 16 processors, I found Cosmos has prediction accuracies of 62% to 93%. I argue that this result justifies more investigation into using prediction to accelerate coherence protocols.

I believe that challenging work lies ahead in high-performance messaging systems. The demand for low-latency communication will continue to grow because as latency drops below certain thresholds new applications are enabled. Unfortunately, latency of communication faces hard physical limits (e.g. speed of light). Fortunately, in the future plenty of cheap computing power, transistors, memory, and disk space will be available. The key question is how these cheap resources can be used effectively to solve the latency problem. One possibility is to use these resources to aggressively initiate *system-wide speculation* in messaging systems.

References

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[3] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, June 1990.

[4] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[5] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 108–120, April 1991.

[6] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles R. Thacker. High Speed Switch Scheduling for Local Area Networks. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 98–108, 1992.

[7] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[8] Gordon Bell. 1995 Observations on Supercomputing Alternatives: Did the MPP Bandwagon Lead to a Cul-de-Sac? *Communications of the ACM*, 39(3):11–15, March 1996.

[9] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, page ?, June 1990.

[10] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 267–284, December 1995.

[11] Matthias A. Blumrich, Cesary Dubnicki, Edward W. Felten, and Kai Li. Protected User-level DMA for the SHRIMP Network Interface. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[12] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[13] Mattias A. Blumrich. Network Interface for Protected, User-Level Communication. Technical report, Department of Computer Science, Princeton University, June 1996. Ph.D. Dissertation.

[14] Mattias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena R. Mesarina. Two Virtual Memory Mapped Network Interface Designs. In *Hot Interconnects II*, 1994.

[15] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[16] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz. Remote Queues: Exposing Message Queues or Optimization and Atomicity. In *Proceedings of the Seventh ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 42–53, 1995.

[17] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D . J. States, S. Swamintathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculation. *Journal of Computational Chemistry*, 4(187), 1983.

[18] Doug Burger and Sanjay Mehta. Parallelizing Appbt for a Shared-Memory Multiprocessor. Technical Report 1286, Computer Sciences Department, University of Wisconsin–Madison, September 1995.

[19] Joseph Carbonaro and Frank Verhoorn. Cavallino: The Teraflops Router and NIC. In *Hot Interconnects IV,* pages 157–160, 1996.

[20] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[21] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[22] Derek Chiou, Boon S. Ang, Arvind, Michael J. Becherle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. StartT-ng: Delivering Seamless Parallel Computing. In *Proceedings of EURO-PAR '95*, Stockholm, Sweden, 1995.

[23] Lynn Choi and Andrew A. Chien. Integrating Networks and Memory Hierarchies in a Multicomputer Node Architecture. In *Proceedings of the Eighth International Parallel Processing Symposium*, 1994.

[24] Fred Chong, Shamik Sharma, Eric Brewer, and Joel Saltz. Multiprocessor Runtime Support for Irregular DAGs. In R. Kalia and P. Vashishta, editors, *Toward Teraflop Computing and New Grand Challenge Applications*. Nova Science Pulishers, Inc., 1995.

[25] Roy Clark and Knut Alnes. SCI Interconnect Chipset and Adapter: Building Large Scale Enterprise Servers with Pentium Pro SHV Nodes. In *Hot Interconnects IV*, pages 225–228, 1996.

[26] Danny Cohen and Chuck Seitz. *M2M-OCT-SW8: Octal 8-Port Myrinet-SAN Switch Recommended Topologies*, 1997. Available from http://www.myri.com/myrinet/switches/m2m-oct-sw8.html.

[27] R.C. Covington, S. Madala, V. Mehta, J.R. Jump, and J.B. Sinclair. The Rice Parallel Processing Testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 4–11, May 1988.

[28] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[29] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[30] David Culler, Lok Tin Liu, Richard Martin, and Chad Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1), February 1996.

[31] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 164–175, Santa Clara, California, 1991.

[32] R. Cypher, A. Ho, S. Konstatinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, 1993.

[33] Donglai Dai and Dhabaleswar K. Panda. How Much Does Network Contention Affect Distributed Shared Memory Performance? In *Proceedings of the 1997 International Conference on Parallel Processing*, 1997.

[34] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Nuth, Scott Wills, Paul Carrick, and Greg Flyer. The J-

Machine: A Fine-Grain Concurrent Computer. In G. X. Ritter, editor, *Proc. Information Processing 89*. Elsevier North-Holland, Inc., 1989.

[35] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *SIGCOMM '94*, pages 2–13, August 1994.

[36] Dave Dunning and Greg Regnier. The Virtual Interface Architecture. In *Hot Interconnects V*, pages 47–58, 1997.

[37] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[38] Bob Felderman. Personal Communication, March 1997.

[39] Marco Fillo, Stephen W. Kekler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. Technical Memo A.I. Memo No. 1532, MIT, March 1995.

[40] Richard M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[41] Mike Galles. The SGI Spider Chip. In *Hot Interconnects IV*, pages 141–146, 1996.

[42] Mike Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, 1994.

[43] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[44] Richard B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[45] Silicon Graphics. Silicon Graphics and Cray Research Unveil Modular Origin Server Family. Available from http://www.sgi.com/Headlines/1996/October/originserver_release.html.

[46] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[47] John Heinlein, Kourosh Gharachorloo, Scott A. Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural*

Support for Programming Languages and Operating Systems (ASPLOS VI), pages 38–50, 1994.

[48] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 274–285, 1994.

[49] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[50] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 111–122, October 1992.

[51] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in it Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V).

[52] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[53] James C. Hoe and Mike Ehrlich. StarT-JR: A Parallel System from Commodity Technology. Computation Structures Technical Memo 384, MITLCS, Oct 1996.

[54] Robert W. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, 15(1):37–45, February 1994.

[55] SPARC International Inc. History of SPARC systems:- the first decade 1987-1996. Available from http://www.sparcproductdirectory.com/history.html.

[56] Myricom Incorporated. Myricom Home Page. Available from http://www.myri.com.

[57] Teresa L. Johnson and Wen mei Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, 1997.

[58] Vijay Karamcheti and Andrew A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 298–307, 1995.

[59] Jonathan Kay and Joesph Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *SIGCOMM93*, pages 259 – 268, 1993.

[60] Kimberly A. Keeton, Thomas E. Anderson, and David A. Patterson. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Hot Interconnects III*, 1995.

[61] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[62] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, 1993.

[63] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 274–284, 1992.

[64] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[65] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[66] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, 1997.

[67] Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, June 1995.

[68] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1993.

[69] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of the Stanford DASH Multiprocessor. Technical Report CSL-TR-89-403, Computer System Laboratory, Stanford University, December 1989.

[70] Lok Tin Liu and David E. Culler. Evaluation of the Intel Paragon on Active Message Communication. In *Proceedings of Intel Supercomputer Users Group Conference*, June 1995.

[71] Tom Lovett and Rusell Clap. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, 1996.

[72] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. Fugu: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor. Technical Memo MIT/LCS/TM-503, MIT Laboratory for Computer Science, October 1994.

[73] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Walter Lee, Anant Agarwal, and M. Frans Kaashoek. UDM: User Direct Messaging for General-Purpose Multiprocessing. Technical Memo 556, MIT Laboratory for Computer Science, March 1996.

[74] Alan Mainwaring and David Culler. Active Messages Applications Programming Interface and Communication Subsystem Organization. Draft Technical Report, Computer Science Department, University of California at Berkeley.

[75] Evangelos P. Markatos and Manolis G. H. Katevenis. User-Level DMA without Operating System Kernel Modification. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, 1997.

[76] Richard Martin. HPAM: An Active Message Layer for a Network of HP Workstations. In *Hot Interconnects II*, 1994.

[77] Robert McMillan. New Ultra 30 workstations signal end of SBus. Available from http://www.sun.com/sunworldonline/swol-07-1997/swol-07-ultra30.html.

[78] Meiko World Inc. Computing Surface 2: Overview Documentation Set, 1993.

[79] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[80] Maged M. Michael, Ashwini K. Nanda, Beng-Hong Lim, and Michael L. Scott. Coherence Controller Architecture for SMP-Based CC-NUMA Multiprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 219–228, 1997.

[81] Sun Microsystems Incorporated. *Rev. 5/20/97 - Workgroup, Dept., & Data Center Servers*, 1997.

[82] Sun Microsystems Incorporated. *Rev. 5/20/97 - Workstation & Workgroup Servers*, 1997.

[83] MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual*, 1995.

[84] Todd. C Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[85] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[86] Shubhendu S. Mukherjee and Mark D. Hill. An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 64–74, Manchester, England, July 1994.

[87] Shubhendu S. Mukherjee and Mark D. Hill. A Case for Making Network Interfaces Less Peripheral. In *Hot Interconnects V*, 1997.

[88] Shubhendu S. Mukherjee and Mark D. Hill. A Survey of User-Level Network Interfaces for System Area Networks. Technical Report 1340, Computer Sciences Department, University of Wisconsin–Madison, February 1997.

[89] Shubhendu S. Mukherjee and Mark D. Hill. The Impact of Data Transfer and Buffering Alternatives on Network Interface Design. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture,* pages 207–218, February 1998.

[90] Shubhendu S. Mukherjee and Mark D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[91] Shubhendu S. Mukherjee, Alain Kagi, and Douglas Burger. A Programming Tutorial for the Wisconsin Wind Tunnel. unpublished manuscript, revised January 1995.

[92] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[93] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[94] Howard M. Needham. Peripheral Component Interconnect (PCI) Bus for ASIC Designers. Available from http://www.ti.com/sc/docs/asic/srga013/toc.htm.

[95] Andreas G. Nowatzyk and Paul R. Prucnal. Are Crossbars Realy Dead? The Case for Optical Multiprocessor Interconnect Systems. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 106–115, 1995.

[96] Randy Osborne, Qin Zheng, John Howard, Ross Casley, and Doug Hahn. DART - A Low Overhead ATM Network Interface Chip. In *Hot Interconnects*, 1996.

[97] Robert W. Pfile. Typhoon-Zero Implementation: The Vortex Module. Technical report, Computer Sciences Department, University of Wisconsin–Madison, 1995.

[98] Ken Polsson. Chronology of Events in the History of Microcomputers. Available from http://www.islandnet.com/ kpolsson/comphist.htm.

[99] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[100] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[101] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[102] Cray Research. Cray Research Redefines Scalable Computing With Cray T3E System, World’s First Truly Scalable Supercomputer. Available from http://www.cray.com/news/9511/scalable.html.

[103] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lucas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, and David A. Wood. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[104] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[105] Ioannis Schoinas and Mark D. Hill. Address Translation Mechanisms in Network Interfaces. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 219–230, February 1998.

[106] Steve Scott and Gregory M. Thorson. The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus. In *Hot Interconnects IV*, pages 147–156, 1996.

[107] Steve L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 26–36, 1996.

[108] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[109] Toshi Shimizu. Personal Communication, June 1997.

[110] Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Low-Latency Message Communication Support for AP1000. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 288–297, 1992.

[111] O. Shiraki, M. Nagatsuka, T. Horie, Y. Koyanagi, T. Shimizu, and H. Ishihata. AP-Net Advanced High-Performance Network for Scalable Parallel Server. In *Hot Interconnects IV*, 1996.

[112] Ashok Singhal, David Broniarczyk, Fred Ceraukis, Jeff Price, Leo Yuan, Chris Cheng, Drew Doblar, Steve Fosth, Nalini Agarwal, Kenneth Harvey, Erik Hagersten, and Bjorn Liencres. Gigaplane (TM): A High Performance Bus for Large SMPs. In *Hot Interconnects IV*, pages 41–52, 1996.

[113] Jonas Skeppstedt and Per Stenstr"om. Simple Compiler Algorithms to Reduce Ownership Overhead in Cache Coherence Protocols. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 286–296, San Jose, California, 1994.

[114] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148, 1981.

[115] James E. Smith and Andrew R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[116] IEEE Computer Society. *IEEE Stanard for Scalable Coherent Interface (SCI)*, 1992.

[117] Gurindar Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

[118] Gurindar S. Sohi and Manoj Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 53–62, 1991.

[119] SPARC Technology Business. *UltraSPARC-I User's Manual, Revision 1.0*, September 1995.

[120] Per Stenstrom, Mats Brorsson, and Lars Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[121] David Stodolsky, J. Bradley Chen, and Brian N. Bershad. Fast Interrupt Priority Management in Operating System Kernels. In *Usenix Microkernels Workshop*, 1993.

[122] Craig B. Stunkel, Dennis G. Shea, Bulent Abali, Mark Atkins, Carl A. Bender, Don. G. Grice, Peter H. Hochschild, Douglas J. Joseph, Ben. J. Nathanson, Richard A. Swetz, Robert F. Stucke, Michael Tsao, and Philip R. Varker. The SP2 Communication Subsystem. *IBM System Journal*, 34(2):185–204, 1995.

[123] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[124] Thinking Machines Corporation. The Connection Machine CM-5 Technical Summary, 1991.

[125] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 191–202, 1997.

[126] Pete Vogt. Profusion: A Buffered, Cache Coherent Crossbar Switch. In *Hot Interconnects V*, pages 87–96, 1997.

[127] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 40–53, December 1995.

[128] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[129] Kai Wang and Manoj Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In *30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 30)*, pages 281–290, 1997.

[130] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu, Thomas Wicki, and Winfried Wilcke. The Mercury Interconnect Architecture: A Cost-effective Infrastructure for High-Performance Servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 98–107, 1997.

[131] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 243–256, April 1989.

[132] Shlomo Weiss and James E. Smith. *Power and PowerPC*. Morgan Kaufmann Publishers, Inc., 1994.

[133] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Hot Interconnects V*, 1997.

[134] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.

[135] David Wood. Personal Communication, April 1998.

[136] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993. Also appeared in it CMG Transactions,/ Spring 1994.

[137] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.

[138] Yong Yao. AGP Speeds 3D Graphics. *Microprocessor Report*, 10(8):11–15, June 17 1996.

[139] T-Y Yeh and Yale Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, 1992.

[140] Albert Yu. The Future of Microprocessors. *IEEE Micro*, 16(6):46–53, December 1996.

[141] Bob Zak. Personal Communication, March 1997.

Appendix A

System Area Networks

There is a new generation of networks that falls somewhere between commercial local area networks (LANs) and custom memory buses. Some of these networks are simply better LANs, some are interconnection networks for massively parallel processors (MPPs), while others resemble high-performance memory buses. These networks are called *System Area Networks* (SANs). To the best of my knowledge, the term “System Area Network” was first used by Robert Horst to describe the Tandem ServerNet.

Today’s conventional LANs are highly scalable and reusable. They connect hundreds of host nodes and provide network interfaces that can be attached to standard I/O buses. However, they do not offer very high performance. Today’s state-of-the-art LANs, such as 100 megabits/second Ethernet or 155 megabits/second switched ATMs, offer very high latency (100-1000s of microseconds) and relatively low bandwidth (10-200 megabits/second). The poor performance of LANs is aggravated by heavy-weight legacy protocol stacks, such as TCP/IP. Such protocols make the conservative assumption that LANs are an extension of the internet and therefore are highly unreliable and able to drop, corrupt, replay, expose, forge, and delay network messages. These assumptions result in complex software protocol stacks that ensure reliability in software.

Memory buses are in striking contrast with LANs. Memory buses deliver extremely low latency (10s of nanoseconds) and very high bandwidth (4 - 20 gigabits/second). Memory buses can be accessed from processors in a few processor cycles because their high reliability and highly trusted environment avoid software intervention. Nevertheless, unlike LANs, memory buses are often customized, have non-standard interfaces, and are hard to extend to hundreds of hosts.

An ideal network would be one that combines the best of memory buses and LANs. Such a network would combine the performance and reliability of a memory bus and avoid running TCP/IP. But the scalability and standardized interfaces of LANs are also desirable so that they can be reused across several generations of machines and/or manufactured by third party vendors. Thus, four goals have given rise to a new generation of networks called System Area Networks (SANs). These goals are:

- Performance (low latency and high bandwidth)
- Reliability
- Scalability and
- Reusability.

Some MPP networks such as the TMC CM-5 network or the Meiko CS2 network can be classified as SANs. More recent examples of SANs are the Myricom Myrinet switch, the IBM Vulcan switch used in SP2, the Spider switch used in the SGI/Cray Origin machine, the Cray T3E network, Dolphin SCI switch, the Fujitsu AP-Net, and the Cray Gigaring. Tandem’s ServerNet can also be classified as a SAN. However, the Tandem ServerNet is unique because it replaces the memory bus, the I/O bus, and the LAN with a single interconnection network called the ServerNet.

Most of these SAN switches deliver latencies of less than a few microseconds and link bandwidth exceeding one gigabit/second. They are highly reliable. They do not drop network messages, provide CRC checks for error detection, and are expected to operate in a closed, secure, and trusted environment such as a business office or a machine room. As a result, errors are extremely rare. If the system does detect a SAN error (e.g., cyclic redundancy check error) it can either crash or return the error status to the user application. SAN switches can be composed to build configurations that connect hundreds of host nodes, which makes them highly scalable (like switched LANs). Finally, SANs provide internal network interfaces that can be reused across different machines or across different generations of the same machine.