Design and Implementation of a Multi-purpose Cluster System Network Interface Unit

by

Boon Seong Ang

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999

Author . . . .
Department of Electrical Engineering and Computer Science
February 23, 1999

Certified by . . . .
Arvind
Johnson Professor of Computer Science
Thesis Supervisor

Certified by . . . .
Larry Rudolph
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
A. C. Smith
Chairman, Departmental Committee on Graduate Students

Design and Implementation of a Multi-purpose Cluster System Network Interface Unit

by

Boon Seong Ang

Submitted to the Department of Electrical Engineering and Computer Science
on February 23, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Today, the interface between a high speed network and a high performance computation node is the least mature hardware technology in scalable general purpose cluster computing. Currently, the one-interface-fits-all philosophy prevails. This approach performs poorly in some cases because of the complexity of modern memory hierarchy and the wide range of communication sizes and patterns. Today's message passing NIU's are also unable to utilize the best data transfer and coordination mechanisms due to poor integration into the computation node's memory hierarchy. These shortcomings unnecessarily constrain the performance of cluster systems.

Our thesis is that a cluster system NIU should support multiple communication interfaces layered on a virtual message queue substrate in order to streamline data movement both within each node as well as between nodes. The NIU should be tightly integrated into the computation node's memory hierarchy via the cache-coherent snoopy system bus so as to gain access to a rich set of data movement operations. We further propose to achieve the goal of a large set of high performance communication functions with a hybrid NIU micro-architecture that combines custom hardware building blocks with an off-the-shelf embedded processor.

These ideas are tested through the design and implementation of the StarT-Voyager NES, an NIU used to connect a cluster of commercial PowerPC based SMP's. Our prototype demonstrates that it is feasible to implement a multi-interface NIU at reasonable hardware cost. This is achieved by reusing a set of basic hardware building blocks and adopting a layered architecture that separates protected network sharing from software visible communication interfaces. Through different mechanisms, our 35MHz NIU (140MHz processor core) can deliver very low latency for very short messages (under 2μs), very high bandwidth for multi-kilobyte block transfers (167 MBytes/s bi-directional bandwidth), and very low processor overhead for multi-cast communication (each additional destination after the first incurs 10 processor clocks).

We introduce the novel idea of supporting a large number of virtual message queues through a combination of hardware Resident message queues and firmware emulated Non-resident message queues. By using the Resident queues as firmware controlled caches, our implementation delivers hardware speed on the average while providing graceful degradation in a low cost implementation.

Finally, we also demonstrate that an off-the-shelf embedded processor complements custom hardware in the NIU, with the former providing flexibility and the latter performance. We identify the interface between the embedded processor and custom hardware as a critical design component and propose a command and completion queue interface to improve the performance and reduce the complexity of embedded firmware.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science

Thesis Supervisor: Larry Rudolph
Title: Principal Research Scientist

Design and Implementation of a Multi-purpose Cluster System Network Interface Unit

by

Boon Seong Ang

Submitted to the Department of Electrical Engineering and Computer Science
on February 23, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Today, the interface between a high speed network and a high performance computation node is the least mature hardware technology in scalable general purpose cluster computing. Currently, the one-interface-fits-all philosophy prevails. This approach performs poorly in some cases because of the complexity of modern memory hierarchy and the wide range of communication sizes and patterns. Today's message passing NIU's are also unable to utilize the best data transfer and coordination mechanisms due to poor integration into the computation node's memory hierarchy. These shortcomings unnecessarily constrain the performance of cluster systems.

Our thesis is that a cluster system NIU should support multiple communication interfaces layered on a virtual message queue substrate in order to streamline data movement both within each node as well as between nodes. The NIU should be tightly integrated into the computation node's memory hierarchy via the cache-coherent snoopy system bus so as to gain access to a rich set of data movement operations. We further propose to achieve the goal of a large set of high performance communication functions with a hybrid NIU micro-architecture that combines custom hardware building blocks with an off-the-shelf embedded processor.

These ideas are tested through the design and implementation of the StarT-Voyager NES, an NIU used to connect a cluster of commercial PowerPC based SMP's. Our prototype demonstrates that it is feasible to implement a multi-interface NIU at reasonable hardware cost. This is achieved by reusing a set of basic hardware building blocks and adopting a layered architecture that separates protected network sharing from software visible communication interfaces. Through different mechanisms, our 35MHz NIU (140MHz processor core) can deliver very low latency for very short messages (under 2μs), very high bandwidth for multi-kilobyte block transfers (167 MBytes/s bi-directional bandwidth), and very low processor overhead for multi-cast communication (each additional destination after the first incurs 10 processor clocks).

We introduce the novel idea of supporting a large number of virtual message queues through a combination of hardware Resident message queues and firmware emulated Non-resident message queues. By using the Resident queues as firmware controlled caches, our implementation delivers hardware speed on the average while providing graceful degradation in a low cost implementation.

Finally, we also demonstrate that an off-the-shelf embedded processor complements custom hardware in the NIU, with the former providing flexibility and the latter performance. We identify the interface between the embedded processor and custom hardware as a critical design component and propose a command and completion queue interface to improve the performance and reduce the complexity of embedded firmware.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science

Thesis Supervisor: Larry Rudolph
Title: Principal Research Scientist

Acknowledgments

This dissertation would not have been possible without the encouragement, support, patience and cooperation of many people. Although no words can adequately express my gratitude, an acknowledgement is the least I can do.

First and foremost, I want to thank my wife, Wee Lee, and our families for standing by me all these years. They gave me the latitude to seek my calling, were patient as the years passed but I was no closer to enlightenment, and provided me a sanctuary to retreat to whenever my marathon-like graduate school career wore me thin. To you all, my eternity gratitude.

I am greatly indebted to my advisors, Arvind and Larry, for their faith in my abilities and for standing by me throughout my long graduate student career. They gave me the opportunity to co-lead a large systems project, an experience which greatly enriched my systems building skills. To Larry, I want to express my gratitude for all the fatherly/brotherly advice and the cheering sessions in the last leg of my graduate school apprentice-ship. I would also like to thank the other members of my thesis committee: Frans and Anant, for helping to refine this work.

I want to thank Derek Chiou for our partnership through graduate school, working together on Monsoon, StarT, StarT-NG and StarT-Voyager. I greatly enjoy bringing vague ideas to you, and jointly developing them into well thought out solutions. This work on StarT-Voyager NES is as much yours as it is mine. Thank you too for the encouragement and counselling you gave me all these years.

The graduate students and staff in Computation Structures Group gave me a home away from home. Derek Chiou, Alex Caro, Andy Boughton, James Hoe, RPaul Johnson, Andy Shaw, Shail Aditya Gupta, Xiaowei Shen, Mike Ehrlich, Dan Rosenband and Jan Maessen, thank you for the company in this long pilgrimage through graduate school. It was a pleasure working with all of you bright, hardworking, devoted and, at one time, idealistic people. I am also indebted to many of you, especially Derek and Alex, for coming to my rescue whenever I painted myself into a corner.

In my final years at MIT, I had the pleasure of working with the StarT-Voyager team: Derek, Mike, Dan, Andy Boughton, Jack Constanza, Brad Bartley, and Wing-chung Ho. It was a tremendous education experience laboring along-side the other talented team members, especially Derek, Mike and Dan. I also want to thank our friends at IBM: Marc, Pratap, Eknath, Beng Hong, and Alan, for assisting us in the StarT-Voyager project.

To all of you I mentioned above, and many others that I missed, thank you for the lessons and memories of these years.

Contents

Chapter 1

Introduction

Today, the interface between a high speed network and a high performance computation node is the least mature hardware technology in scalable general purpose cluster computing. Currently, the one-interface-fits-all philosophy prevails. This approach performs poorly in some cases because of the complexity of modern memory hierarchy and the wide range of communication sizes and patterns. Today's message passing NIU's are also unable to utilize the best data transfer and coordination mechanisms due to poor integration into the computation node's memory hierarchy. These shortcomings unnecessarily constrain the performance of cluster systems.

Our thesis is that a cluster system NIU should support multiple communication interfaces layered on a virtual message queue substrate in order to streamline data movement both within each node as well as between nodes. The NIU should be tightly integrated into the computation node's memory hierarchy via the cache-coherent system bus so as to gain access to a rich set of data movement operations. We further propose to achieve the goal of a large set of high performance communication functions with a hybrid NIU micro-architecture that combines a set of custom hardware basic building blocks with an off-the-shelf embedded processor.

Together, these features provide a cost effective solution for running *mixed workloads* encompassing parallel, distributed client-server, and sequential applications. This means achieving both good overall *system utilization* and high *single application performance* across applications with widely *different communication requirements*.

Our proposed NIU architectural features address the seemingly opposing requirements of high performance, multiple communication interface support while catering to system-level issues of sharing, protection, and job scheduling flexibility.

1.1 Motivation

Connecting a cluster of commercial Symmetric Multi-processors with a high performance network is an attractive way of building large general purpose computation platforms. By exploiting existing high volume commercial hardware and software as the building blocks, this approach both reduces cost and provides an evolutionary system upgrade path. It also offers other advantages such as modular expansion and, with appropriate system software, higher availability because of multiple instances of similar resources.

The communication system is a key component of a cluster system and should have the following attributes. Realizing these goals will require support in the NIU.

- A general purpose cluster system will encounter applications with a range of communication requirements, *e.g.* fine and coarse grain communication, and shared memory and message passing styles of communication. For both compatibility and performance reasons, it is important that a cluster system supports a range of communication interfaces.

- High performance is important for the system to be able to support fine grain parallel processing and aggressive resource sharing across the cluster. Ideally, inter-node latency and bandwidth should be no more than a few times worse than those within a node.

- Finally, system oriented support – efficient network sharing, full communication protection, and flexible job scheduling – is also critical as system throughput is just as important as single application performance.

Today's NIU's, divided between the two camps of message passing NIU's and shared memory NIU's, fall short of these goals.

All NIU's today are designed with a one-interface-fits-all philosophy, presumably to keep hardware simple and fast. Should an application desire a communication interface different from that provided by the underlying hardware, software layers are used to "synthesize" it from the hardware supported operations. Unfortunately, neither types of NIU offer a "communication instruction set" sufficient for efficient software synthesis of all common communication interfaces.

The problem is particularly acute for message passing NIU's as it is very difficult to synthesize shared memory in a way that is transparent to application programs. A number of methods have been attempted [67, 53, 95, 94], but none have found wide acceptance. With the growing popularity of SMP and SMP applications, this is a significant draw back.

Shared memory NIU's have fared better, since it is functionally simple to synthesize message passing on shared memory. Nonetheless, shared memory emulated message passing incurs more network trips than a direct implementation. This points to an interesting feature of shared memory systems – software has no direct control over data movement, which occurs indirectly in response to cache-misses. Although it simplifies programming, this feature also creates inefficiency, particularly for control oriented communication.

The work of Mellor-Crummey and Scott [74, 75] on shared memory implementations of mutex lock and barrier provides an interesting illustration. At a meta-level, the solution is to understand the behavior of the underlying coherence protocol, and craft algorithms that coax it into communicating in a way that is close to what a direct message passing implementation would have done. Unfortunately, due to the lack of direct control over communication, even these clever lock and barrier implementations incur more network traffic than a message passing implementation of similar algorithms (See Heinlein [40][1]).

[1]Heinlein's implementation is best viewed as a hybrid that blurs the line between message passing and shared memory. A large part of the message passing code implementing the lock and barrier runs on an embedded processor in the NIU, with application code interacting with this code through memory mapped interfaces. One could either view this as extending shared memory with special lock and barrier protocol, or as message passing code off-loaded onto the NIU.

The bottom line is that shared memory system's communication instruction set is also inadequate, resulting in redundant data movement between nodes. Interestingly, while message passing NIU's provide good control over data movement between nodes, many of them are not well integrated into the node's memory hierarchy resulting in inefficient data movement and control exchange within node. Solving this problem requires the NIU to be located on the cache-coherent system bus so that it has access to a rich set of intra-node data movement operations. In addition, the NIU should offer multiple communication interfaces so that software has a sufficiently rich communication instruction set to synthesize its desired communication efficiently.

Supporting multiple communication interfaces requires re-visiting the issue of network sharing and protection. Traditionally, message passing and shared memory systems handle this issue differently. Shared memory machines essentially skirt this issue by providing shared communication access without allowing direct network access. Although entities in different protection domains can communicate concurrently through shared memory accesses – with protection enforced by normal virtual address translation mechanism – the fast network is used directly only by cache-coherence protocol traffic.

Message passing systems that permit direct user-level network access have to resolve a tougher network sharing protection problem. Aside from preventing illegal message transmission and reception, the protection scheme has to prevent deadlock and starvation that may arise from sharing network resources between otherwise independent jobs. In our opinion, this problem has never been solved satisfactorily in existing systems. Current solutions, discussed in Section 2.3.2, suffer from one or more draw-backs including harsh restrictions on job scheduling policies, significant latency penalty, and constrained functionality.

Network sharing and protection get even more complex when both shared memory and message passing interfaces are supported over the same network. Interaction between the two breaks some solutions used in systems that support only one of them. For example, a solution employed in message passing systems to prevent network deadlocks is to rely on software to continually accept packets from the network. In

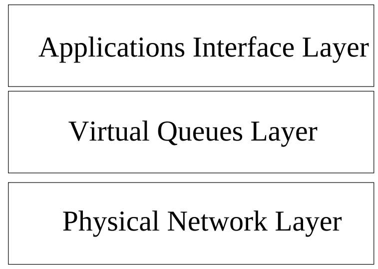


Figure 1-1: Our proposed layered cluster system communication architecture.

a system with both shared memory and message passing support, this no longer works because software may not get a chance to service messages: its processor may be stalled waiting for a cache-miss processing to complete. In turn, the cache-miss processing could be waiting for network resources held by message passing traffic to become available. Solving problems of this nature requires a comprehensive protected network sharing model.

1.2 Proposed Cluster Communication Architecture

We propose a layered communication architecture illustrated in Figure 1-1 to meet the complex communication requirements of a cluster system. This design assumes that the NIU resides on the SMP node's memory bus, giving it the ability to participate in cache-coherence snooping operations. It also assumes an NIU with a programmable core, making it feasible to support a large and extendable set of communication functions.

The *Physical Network Layer* is a transport layer that provides reliable packet delivery over two logically independent networks. It is also responsible for regulating the flow of traffic through the network to avoid network congestion.

The *Virtual Queues Layer* implements the bulk of the protection scheme in our system. It operates on the abstraction of virtual message queues, using the packet transport services provided by the Physical Network layer to move each message from its transmit queue (TxQ) to its receive queue (RxQ). By controlling local queue access and transmit-to-receive-queue connectivity, the Virtual Queues layer provides system software with the mechanism to "stitch" virtual message queues into independent communication domains.

The Virtual Queues layer deals with the problem of network deadlock arising from dynamic receive queue buffer space allocation with a novel *Reactive Flow-control* scheme. This lazy queue-to-queue flow-control strategy incurs no flow-control cost when communication traffic is well-behaved, imposing flow-control only when a possible problem is detected. (See Section 3.2.3 for further details.)

Virtualization of message queue name in the Virtual Queues layer introduces a level of indirection which facilitates job migration and the contraction of the number of processors devoted to a parallel job. Furthermore, our design allows a message queue to remain active, independent of the scheduling state of the process using it. Together, these features give system job scheduling unprecedented flexibility.

Finally, the *Application Interface Layer* focuses on the interface seen by application code. One could view this layer as providing "wrappers" around the message queue service of the Virtual Queues layer to form the communication instruction set. As an example, a wrapper for large messages marshals data from, and stores data into user virtual address space, implementing virtual memory to virtual memory copy across the network. Another wrapper in our design implements an interface crafted to reduce end-to-end latency and message send/receive handshake overhead of very short messages.

Cache-coherent distributed shared memory support can be viewed as yet another wrapper, albeit a sophisticated one. This wrapper observes transactions on the memory bus, and where necessary, translates them into request messages to remote nodes. It also services requests from other nodes, supplying data, initiating invalidation or update actions, or executing these actions on local caches to maintain data coherence in the shared address space.

It is feasible for an NIU to support a fair number of wrappers because although they export different communication abstractions, their implementations share many subcomponents. As such, the actual provision of multiple Application Interface layer abstractions can be achieved with a relatively small set of *composable primitives*.

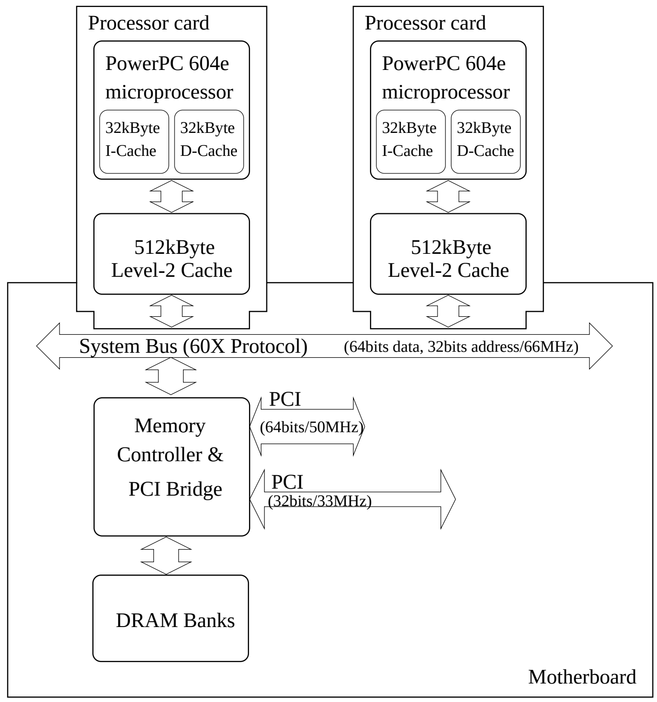
By dividing the communication architecture into several layers with well defined responsibilities, this approach not only avoids the duplication of functions, but also simplifies the internal design of each layer. For instance, with the Virtual Queues Layer responsible for network sharing issues, the design and implementation of each instance in the interface layer can be done in isolation, without concern about how other instances are using the shared network.

1.3 A Flexible Network Interface Unit (NIU) Design

The abstract communication architecture described in the previous section is tested in an actual implementation, the StarT-Voyager Network Endpoint Subsystem (NES). The NES connects a cluster of IBM PowerPC 604e-based SMP's[2] to the Arctic network [12, 13], a high performance, packet switched, Fat-Tree [63] network. Figure 1-2 illustrates how the StarT-Voyager NES replaces one of the processor cards in the normal SMP to interface directly to the SMP's cache-coherent 60X system bus. The diagram also shows the two main components of the NES: an NES Core containing custom logic, and an sP subsystem containing a PowerPC 604 processor used an an embedded processor. We refer to this processor as the Service Processor (sP),and the host SMP's processors as the Application Processors (aP's).

The NES micro-architecture attempts to achieve both performance and flexibility through the combination of custom hardware and embedded processor firmware. By ensuring that the common communication operations are fully supported in hardware, average performance is close to "hardware speed". On the other hand, the sP firmware handles the infrequent corner cases, and provides extension capabilities. This combination ensures that the hardware can be kept simple and thus fast, despite

[2]These are the IBM RISCSystem/6000 Model 43P-240 machines, first introduced in the fourth quarter of 1996.



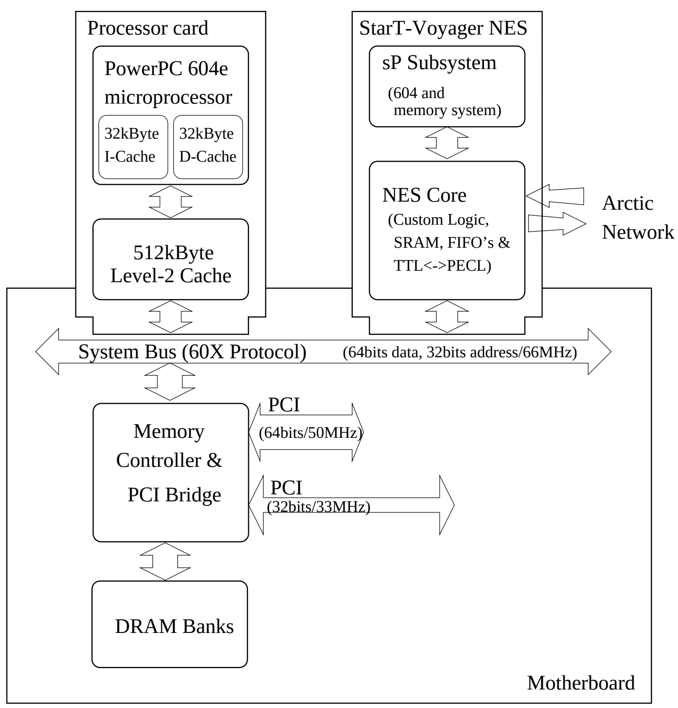


Figure 1-2: The top of the diagram shows an original SMP. The bottom shows a one used in the StarT-Voyager system, with a processor card replaced by the StarT-Voyager NES.

the NES's support for a large set of functionalities. Key to the effectiveness of this design is the interface between the sP and the NES Core, described in Section 4.4.

Implementation of multiple virtual queues provides a concrete example of this hybrid hardware/firmware implementation approach. The abstraction of a large number of active message queues is achieved with a small number of NES Core implemented hardware *Resident* queues and a large number of sP firmware implemented *Non-resident* queues. Both types of queues export identical interfaces to aP software. In addition, the Resident queues can be used as system software (or sP firmware) managed caches of the Non-resident queues, and switching a queue between Resident and Non-resident resources is transparent to code using the queue.

The NES Core is designed as a collection of communication primitives. These are assembled by other NES custom hardware or sP firmware into the functions available to aP software, providing an efficient way to implement multiple abstractions while facilitating future extension. As an illustration of this design concept, the NES Core supports several different message passing mechanisms – Express, Basic, Express-TagOn, Basic-TagOn and inter-node DMA – catering to messages of increasing size.

Although coherent shared memory is not a focus of this work, the NES Core provides sufficient hardware hooks so that sP firmware can implement various cache coherence protocols. This part of the design emphasizes flexibility for experimentation over absolute performance. With researchers constantly coming up with suggestions for modifying cache coherence protocol to improve performance (*e.g.* [23, 29, 51]), we believe that it is useful to have a platform which allows easy modification to its protocol so that meaningful comparison with real work-load can be done.

The StarT-Voyager NES is designed to facilitate migration of performance critical portions of sP firmware into hardware. Major portions of the StarT-Voyager NES are implemented in FPGA's, with the base design occupying less than a third of the total FPGA space. Because the StarT-Voyager NES is designed in a modular fashion with well-defined functional units, moving a function from sP firmware to custom hardware involves the addition of a new functional unit without major perturbations to the existing design. With two levels of programmability, StarT-Voyager allows new functions to be implemented and fully debugged in firmware first; subsequently, portions that are critical to performance can be moved into FPGA hardware.

1.4 Related Work

This section compares our NIU design with some closely related work. We leveraged many of the ideas proposed in these work. Section 2.5 describes NIU design in general, and includes references to a larger set of related work.

SynfinityNUMA

A recent commercial product, the Fujitsu SynfinityNUMA [102], is very similar to our design in that it employs a layered NIU architecture, with each layer handling a different function. Like our communication architecture, the top layer deals with issues of interfacing to the applications. Their second layer is responsible for recovering from packet losses, and as such presents a level of service equivalent to our bottom layer. Their third layer provides a raw network transport service that has low but non-negligible loss/error rate. They do not have a layer corresponding to our Virtual Queues layer. The network interface in SynfinityNUMA is not programmable. Hence, although it supports both message passing and shared memory, there is no capability for any further communication interface extensions.

FLASH

The FLASH [59, 60] multiprocessor's communication element, the MAGIC chip, is the closest to our design from a capability perspective. A custom designed chip, MAGIC interfaces to the node processor, DRAM, PCI I/O bus and a high performance network. It has a programmable core, called the PP, which coordinates data movement between the four components that it interfaces with. The PP also runs cache-coherence protocol code. Because MAGIC occupies an extremely strategic position in the compute node and is programmable, it can be programmed to implement any function that our design can offer.

Our design differs from MAGIC/FLASH in that we want to interface to commercial SMP's without replacing their memory controller and I/O bus bridge chip. As a result, we face different design constraints. In that respect, DASH [65, 66], the predecessor of FLASH, is more similar but it only supports shared memory and has no programmability.

Our NIU design also takes a very different micro-architecture approach. In the FLASH design, all memory, I/O, and network communication is processed by MAGIC's programmable core. Obviously, the performance of this programmable core is critical and their research focuses on making this fast. We are interested in using an off-the-shelf microprocessor as the programmable element in our NIU to lower development cost and capitalize on the rapid improvements and higher core performance of commercial microprocessors. To compensate for the weaknesses of our approach, such as slow off-chip access in today's microprocessors, we provide full NIU hardware support for the most common and simple communication operations. Furthermore, the NIU provides the embedded processor with a rich set of communication oriented primitives. These enable the embedded processor to orchestrate data transfer without directly touching data, and to coordinate a sequence of primitives without constantly monitoring their progress.

As a result of the difference in implementation approach, the programmability of the two designs are good for different things. For performance reasons, our NIU's programmable portion can only be used to service infrequent operations. But it can afford to execute more complex sequences of code than can MAGIC's PP on each occasion.

Typhoon

The Typhoon design [91] is similar to ours in that they advocate using a commercial microprocessor to provide programmability in the NIU. The design mentions support for both message passing and shared memory, though there is no publicly available description of message passing aspect of the machine. No machine beyond the Typhoon-0 [92], a stripped down version of the design, was built.

Typhoon-0 uses Myrinet [11], an I/O bus NIU, to provide message passing service, an existing SMP processor (one of the Sparc processors in their SUN SMP) as protocol engine, and a custom bus snooper hardware which imposes cache-line granularity access permission checks on bus transactions to main memory DRAM. Typhoon-0 has little control over the design of most of its components since they come off-the-shelf. Its design does not address many of the micro-architectural issues that we studied.

Alewife

Alewife [1, 58], an earlier experimental machine built at MIT, is similar to our work in that it supports both message passing and shared memory. Its communication support is programmable in an interesting way. Alewife uses a modified Sparc processor, called Sparcle [2], which has hardware multi-threading and fast interrupt support. This makes it feasible for its CMMU [57], the cache and memory management unit which also serves as the network interface unit, to interrupt Sparcle and have software take over some of its functions. For instance, this is used to handle corner cases in Alewife's cache-coherence protocol [17].

Alewife does not deal with issues of protection and sharing. These are investigated in a follow-on project, FUGU [70, 69, 71]. Aside from adding small hardware extensions to Alewife, FUGU relies on Sparcle's fast interrupt and interrupt software to impose sharing protection.

The constraints we faced are very different from those faced in Alewife and FUGU. For example, Sparcle has no on-chip cache, and Alewife's CMMU essentially interfaces to Sparcle's L1 cache bus. This is much closer to the processor core than any possible point of interface in today's commercial SMP's. Sparcle's fast interrupt and multi-threading support is also not available on today's commercial microprocessors, whose complex superscalar and speculative processing pipelines are slow to respond to interrupts. These differences make many of Alewife's solutions inapplicable to a cluster of unmodified SMP's.

Hamlyn

Hamlyn [16, 15], a message passing interface architecture that was implemented on the Myrinet hardware, shares our work's goal of sharing a fast network between multiple users in a protected fashion without scheduling restriction. Through their choice of requiring the sender to specify the destination memory to write a message into, they avoid one of the major problems of sharing the network – dynamic receive queue buffer allocation. Whereas the NIU is responsible for receive buffer allocation in our architecture, this task is relegated to the message sender software in Hamlyn. Hamlyn shares another similarity with our design: support for several different message types targeting messages of varying granularity.

Remote Queues

Brewer and Chong [14] proposed exposing message receive queues as Remote Queues to software in order to optimize message passing. The abstraction of message queue is a central part of our protection mechanism, and we leveraged their work.

1.5 Contributions

The main contribution of this work is the proposition that the NIU architecture for cluster systems can be endowed with a rich set of capabilities that provide sharing, protection, and flexibility without sacrificing low latency and high performance. This proposition is supported with a full, realistic NIU design and associated simulations. The following is a list of novel features explored in this work:

- A network sharing scheme based on multiple, virtual message queues and a simple Reactive flow-control strategy. This scheme permits flexible network sharing while retaining full protection and low communication latency.

- A cost effective, high performance implementation of the multiple virtual message queue abstraction using caching concepts.

- A set of efficient message passing primitives crafted to achieve high performance over a wide range of message sizes in an SMP environment.

- An NIU micro-architecture that couples a commercial microprocessor with custom hardware to achieve high performance and flexibility. The custom hardware is organized around a basic set of communication primitives assembled by other NIU hardware or embedded processor firmware into multiple communication interfaces.

- The actual construction of an NIU, the StarT-Voyager NES, which embodies most of the ideas proposed in this work.

The primary emphasis of this thesis work is demonstrating the feasibility of an NIU which directly supports multiple, extendable communication interfaces. It does not attempt to provide a definitive study of whether this NIU architecture leads to better performance than alternate communication architectures, such as one that relies only on cache-coherent shared memory hardware. Furthermore, evaluation in this thesis focuses on the message passing aspect of StarT-Voyager NES, as shared memory support is studied by other research group members [99, 98, 97].

Our evaluation of the StarT-Voyager NES shows that with an appropriate architecture, multi-interface support and flexible protected sharing of the NIU and network are compatible with high performance and low latency. For example, a comparison with StarT-X [42], an I/O bus message passing NIU implemented in similar technology but without NIU programmability or protection for network sharing, shows that the StarT-Voyager NES delivers superior message passing performance – higher bandwidth for large transfers, and lower latency for the shortest messages. Part of the advantage derives from being on the system memory bus, while the remainder comes from the multiple message passing mechanisms of StarT-Voyager, each of which has its own "sweet spot".

We also show that only a small amount of fast memory and custom logic is required for virtual queue support. A large number of message queues can be supported using a Resident/Non-resident scheme: only a small number of queues are buffered in the NES while the rest are buffered in DRAM[3]. Although message passing through Non-

[3]Determining the exact size of the message queue "working set" is beyond the scope of this thesis, as it is dependent on node size and workload. In the StarT-Voyager test bed, we provide sixteen

resident queues incurs longer latency and achieves lower throughput, the degradation is reasonable – less than a factor of five in the worst case.

The Non-resident message queues, implemented with firmware on the sP, provide our first examples of the utility of the sP. As further study of the sP, we conducted a set of block transfer experiments (see Section 5.4) with application code, sP code and dedicated hardware taking on varying responsibilities. These experiments show that the sP is functionally extremely flexible. They also show that when pushed to the extreme, sP performance is limited by context switch overhead when servicing fine-grain communication events, and by off-chip access when handling coarse-grain communication events.

1.6 Road Map

Chapter 1 provided a summary of this thesis: why we are interested in this work, the problems addressed, the solution proposed, both abstract and concrete, our contributions and results.

Chapter 2 examines the communication requirements of cluster systems, what is needed from the NIU, current NIU design practices, and why these designs are inadequate.

Chapter 3 presents an abstract three-layered network interface architecture that meets the goals set forth in Chapter 2. In addition to describing each of the three layers, this chapter also explains the rationale behind the design choices, including comparing them with alternate design options.

Chapter 4 describes the StarT-Voyager NES, a concrete implementation of the architecture proposed in Chapter 3. The micro-architecture of the NES is first presented at the functional level. Next, the mapping of the abstract architecture of Chapter 3 onto the micro-architecture is described. Finally the hardware

transmit and sixteen receive resident queues, a number which should be sufficient for the needs of the operating system and the current, previous, and next user jobs.

mapping of the functional blocks into physical devices and the hardware design flow is presented.

Chapter 5 presents quantitative evaluation of the NES. These evaluations are done on a simulator because the NES hardware was not available in time for this work. Using micro-benchmarks, a series of experiments demonstrate the performance of both the fully hardware implemented Resident message queues and the sP implemented Non-resident message queues. A second series of experiments examine several different ways of implementing block transfers on the NES. They not only demonstrate the versatility of the NES's programmability, but also throw light on the performance potential and limits of the design.

Chapter 6 summarizes what we learned from this research, and suggest several avenues for future work.

Chapter 2

Design Requirements

An NIU designed to support multiple communication abstractions and share a fast network between traffic in several protection domains has to overcome a number of challenges. One is implementing the multiple communication abstractions efficiently, *i.e.* achieve good performance while working within reasonable hardware cost and design complexity. A second challenge is sharing the network in a protected fashion without degrading performance such as incurring longer latency. A third issue is achieving efficient interaction between the application processor and the NIU so as to keep communication overhead low; this is particularly challenging for message passing.

This chapter approaches these issues by examining the communication needs of cluster systems, the restrictions imposed by commercial SMP's, and the role of the NIU in meeting these requirements. Sections 2.1 and 2.2 survey the current practices and promising new directions in message passing and shared memory communication respectively. These functions constitute a core set of capabilities that our NIU has to support. Next, Section 2.3 examines system-level communication issues. Since our design targets commercial SMP's as the host nodes, it has to respect SMP imposed restrictions discussed in Section 2.4. The last section of this chapter, Section 2.5, explains what an NIU does to deliver these communication functions. We approach this task in an incremental fashion, and in the process highlight some existing NIU's that are representative of particular classes of NIU.

2.1 Message Passing Communication

Message passing is a broad term referring to communication involving explicit software send and receive actions. In addition to transporting data from a sender to a receiver, message passing often associates control implications with the events of sending or receiving a message. Occasionally, the term is also used to include Get and Put operations – unilateral remote data fetch and write actions explicitly requested by software on one node, but completed without direct participation of software on the remote node.

Application code usually utilizes the message passing service of a library which presents it with a convenient, portable interface. Any mismatch between this interface and the machine's native communication capability is hidden by the library code. As to be expected, a good match between the library's message passing interface and the machine's capability results the cost of library emulation code.

Message passing libraries come in many forms. Some, crafted to be fast, offer slightly abstracted versions of the machine's underlying communication support. Examples include Active Message [100], Fast Message [84] and Intel's Virtual Interface Architecture [28]. Others, such as NX [88], PVM [35] and MPI [31, 27] offer many more functions meant to simplify the task of writing message passing parallel programs. The following are some common variations among the different message passing services.

2.1.1 Channel vs Queues-and-network Model

Two connectivity model: channel and queues-and-network are common among message passing libraries. Under the channel model, each channel connects exactly one sender to one receiver. In contrast to this one-to-one model, the queues-and-network model offers many-to-many connectivity, where each sender can send messages to a large number of destinations using the same transmit queue. Each receiver similarly can receive messages from a large number of senders through one receive queue. Whereas the message destination of each channel is fixed when it is set up,

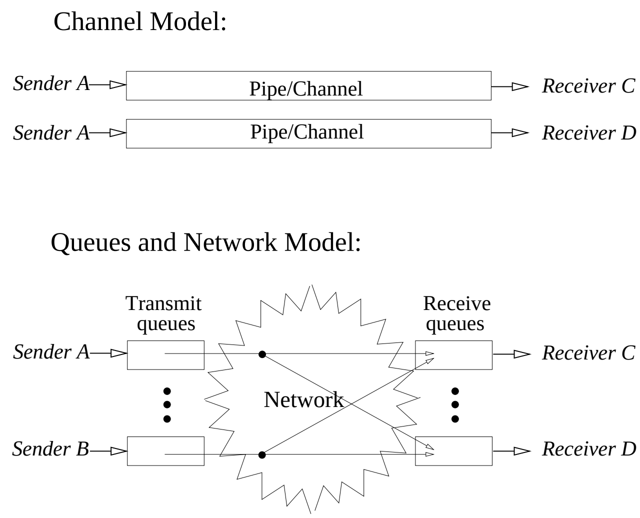


Figure 2-1: Two common models of communication: channels and network connected queues. A channel connects a sender to only one receiver. In contrast, messages can be send to multiple receive queues through each send queue.

each message sent via a transmit queue specifies its destination.

The queues-and-network model is advantageous when the communication pattern is dynamically determined and involves many source-destination pairs. To connect s senders with r receivers, the channel model requires $(s \times r)$ channels, compared to queues-and-network's $(r + s)$ queues. Furthermore, if messages are received by polling, the channel model requires each receiver to poll from up to s channels if the source of the next message is unknown.

The channel model is appropriate when the communication pattern is static and involves only a small number of source-destination pairs. Many traditional parallel programs involve only nearest neighbor communication and global barrier synchronization which can be implemented with a reduction tree. In such cases, channels are only needed between nearest neighbors and neighboring nodes in the reduction tree.

The queues-and-network model is more flexible than the channel model but comes at the price of more complex message buffer management as we will see later in Sections 2.1.3 and 3.2.2. It is our design goal to support this model because it offers a super-set of the functions of the channel model.

2.1.2 Message Content Specification

The content of a message can be specified in one of two ways: (i) by reference, with the starting address and transfer size, or (ii) by value, with the data copied explicitly into some special send buffer. Specification by reference can potentially reduce the number of times message data is copied. Zero copying is possible provided the NIU is able to access memory with user virtual addresses. This means that the NIU must be able to translate user virtual addresses into physical addresses because the host SMP's system bus deals with physical addresses only. The alternative of making system calls to translate virtual addresses to physical addresses incurs unacceptably high overhead. Unfortunately, many NIU's are not equipped with this translation capability so that when a library interface uses specification by reference, the library code ends up copying the message data.

Zero copying is not always an important goal. If the message contains very little data, data copying overhead is small while directly writing the data to the NIU reduces message passing latency. In addition, specification by reference is advantageous only if the location of the data can be specified easily, *e.g.* if the message data already exists in contiguous memory locations or at some regular stride. Otherwise, the task of describing the data layout may be as expensive, if not more expensive than assembling the data into the send buffer. If message content can be specified by reference only, application code may end up assembling data in normal memory location. This nullifies the advantage of specification by reference.

A good design should support specification by value for short to medium sized messages, while adopting specification by reference for large messages.

2.1.3 Message Buffer Management

Message passing requires buffer space for storing messages between a send and the corresponding receive. Buffer management is closely tied to the message passing interface design, with many possible division of responsibilities between the NIU and the software using its message passing service. Furthermore, these choices have implications for network sharing – in a shared network, some buffer management choices are more prone to network deadlocks than others. We begin with a discussion of transmit buffer space management followed by one on receive buffer space management.

Transmit Buffer Management

When transmit buffer space is unavailable, two interface design options are possible: (i) block the sender, or (ii) notify the sender about send failure. Blocking the sender means hardware stalls the sender so that software is unaware of the problem. It is less flexible than the second option because a sender notified of a send failure can not only retry in a tight loop to emulate blocking, but also has the option of carrying out other computation or receive actions before attempting to send again. This added flexibility not only improves performance, but is needed to prevent communication deadlocks.

Although the above paragraph uses the term "notify the sender of send failure", this behavior is typically achieved with software testing for sufficient transmit buffer space before sending a message. The sender could be notified of send failure through an exception/interrupt, but such a design is not used today because of implement difficulty. An exception/interrupt based scheme must provide means for software recovery; at the very least, the exception must occur before software attempts to transmit yet another message. This is not easy to guarantee when the NIU is not integrated into the processor core. The high cost of interrupt/exception handling on most processors and OS's also makes its performance advantage unclear.

Programs that can estimate its maximum transmit buffering requirement can avoid the dynamic buffer space availability check overhead by allocating the maximum required space in the channel or transmit queue. Others that need to deal with the out-of-send-buffer problem must ensure that buffer space will eventually free up to avoid getting into a deadlock. This is a classic resource allocation with dependence problem, and an application must ensure that no cycle forms in the dependence graph.

Receive Buffer Management

Receive buffer management is more complex. It interacts with network sharing, and the detail requirements are dependent on the connectivity model. We first discuss receive buffer management for channel models.

The simplest way to implement the channel model uses the same amount of buffer space in both the sender and receiver, but expose only the buffer space of one side to software. It treats the destination buffer as an eagerly updated mirror of the source buffer. Whenever a message is enqueued into the channel, the source NIU can forward it to the destination with full confidence that there is sufficient destination buffer to accommodate it. The source buffer space provides transient buffering should the network be congested. This approach simplifies NIU design as it never runs into an out-of-buffer situation; that problem is handled by software when it attempts to enqueue into the channel. However, the simplicity comes at the expense of less efficiently buffer space utilization – only half the total amount of buffer space used to implement a channel is exposed to software. Any attempt to expose more of the combined buffer space to user code requires additional coordination between the source and destination NIU.

Receive buffer space allocation in the queues-and-network model is more difficult. Typically, this is done dynamically at the time a message arrives. When an application is unable to estimate the maximum amount of receive queue space needed, a message may arrive at its destination to find its receive queue out of buffer space. Most systems deal with this situation in one of two ways: (i) the message is blocked, *i.e.* continue to occupy the transient buffer in the network, (ii) the message is dropped. A third option of returning the message to the sender is sometimes used [30], but requires appropriate support to ensure that: (a) the sender has space to buffer the returned message before re-sending it again, and (b) there is a logically separate network dedicated to return traffic.

Blocking an incoming message when its destination queue is full causes blockage into the network, which may lead to deadlocks. Nevertheless, the approach has been adopted in some machines, *e.g.* the CM-5, where multiple logically independent networks and a software network usage discipline ensures that the blockage never develops into dependence cycles [64].

If messages are dropped when buffer space is unavailable at the destination, the message loss can either be exposed to user code, or hidden by the message passing library with recovery protocol. It is important to note that dropping the message prevents only those deadlocks due to dependence cycles involving transient, shared, system resources, such as buffers in network switches. The application still has to ensure that there is no dependence cycles involving privately owned resources such as its private receive queue buffer space. Otherwise, dropping the packet merely converts a network deadlock into communication live-lock, where repeated attempts to deliver a message fails with the message dropped at the destination NIU.

The issue of dynamic receive buffer allocation can be "legislated" away with an NIU interface that requires a message sender to specify the destination buffer address for each message. In essence, the interface is equivalent to remote write initiated through a message transmit interface. VIA [28] and Hamlyn [16, 15], for instance, take this approach. However, this approach basically pushes the buffer management duties up one level. If that level is also hardware such as cache-coherence protocol in the NIU, it is unclear that the problem has gotten any easier. As such, it is our goal to support dynamic receive buffer allocation even in the presence of network sharing.

Dynamic buffer allocation and network deadlocks in a shared network environment is a major problem in NIU design, which we revisit in Section 3.2.2. For now, it suffices to say that in a system where network resources are shared by multiple logically independent applications, use of these shared resources have to be done with care to prevent dependences from building up *between* otherwise independent jobs.

2.1.4 Message Reception Method

The most common methods of receiving messages are by polling or via interrupts. Because interrupts are expensive in most systems today, polling is the recommended method for high performance systems when communication is frequent. Interrupts also introduces atomicity issues that force the use of mutex locks which reception by polling avoids. Nevertheless, polling has its inconvenience and overhead. When timely servicing of messages is important, polling code has to be inserted into numerous parts of the receiver's code, making the resulting code difficult to read and debug.

Ideally, both reception by polling and by interrupt should be supported. The receiver should be able to dynamically opt for polling when it expects messages, and for interrupt when messages are expected to be rare. It is also useful for the sender to request that a particular message cause an interrupt, *e.g.*, when the sender urgently needs the receiver's attention.

While messages are typically received by application code, this may be preceded by NIU hardware or firmware pre-processing. An example is the proposed StarT system [80] based on the 88110MP [85] processors which treats each message in the system as a continuation composed of an instruction pointer (IP) and a list of parameters. Designed to be a multi-threaded processor operating in a continuation-passing fashion, the 88110 MP includes an NIU that pre-processes each in-coming continuation message, incorporating it into the local, hardware-supported continuation stack/queue. The processor hardware maintains several priorities of continuations, and supports a branch instruction that jumps to the highest priority continuation.

Processing in the NIU can also completely take care of a message. Examples include Put or Get operations (available on DEC's Memory Channel [36]), and NIU implemented barrier or lock. Aside from freeing the processor to concentrate on executing longer threads, shifting servicing of in-coming messages to the NIU ensures timely servicing; the NIU is always "scheduled", is continually servicing short requests, and does not face suspension due to interrupts like a page fault.

2.1.5 Communication Service Guarantee

Application programmers are usually interested in whether message delivery is (i) reliable, *i.e.* lossless, and (ii) in-order, *i.e.* messages between each source-destination pair arrive in the order sent. Although some applications can tolerate message losses, most require reliable message delivery. In-order delivery is very useful to certain applications as it helps reasoning about distributed events; for instance, cache coherent distributed shared memory protocols that can assume in-order delivery of messages are simpler as fewer scenarios are possible.

Messages may be lost for many reasons. Some networks do not guarantee lossless service. Optical networks for example are designed to operate at high speed with very low, but non-negligible error rates. Networks may also adopt link-level control which drops a message when its destination port fails to make forward progress within a time-out interval (*e.g.* in Myrinet). Finally, some NIU's are designed to drop messages when their destination queues are full; hence, even if the network delivers messages reliably, the NIU may drop them.

Packets sent from one source to the same destination may not arrive in the order sent due to several causes. Some networks provide multiple paths between a source-destination pair of nodes and adaptively routes packets to achieve higher bandwidth or improve fault tolerance. Flow-control protocols can also disrupt ordering if messages are dropped and retried without regard for ordering.

Depending on the network and the NIU designs, providing lossless and in-order delivery guarantees may incur extra overhead in the form of protocols, usually some form of Sliding Window protocol, to recover from losses, re-construct the ordering, or achieve both. A portable message passing library is therefore better off leaving both as options so that applications which can tolerate some losses or out-of-order messages do not incur the corresponding protocol overhead. At the same time, a network and NIU design that delivers these properties without the overhead of these protocols is very useful.

2.1.6 Synchronization Semantics

Message passing send and receive actions often have associated control semantics. A receive action is often used as a mechanism to initiate action at the destination. The information received can also inform the receiver that the sender has reached a particular point in execution.

Some message passing libraries also support *blocking* send and receive actions. In a blocking send, the sender's execution is suspended until the message is received at the destination. In a blocking receive, the receiver's execution is suspended until a message arrives. The alternatives to blocking receive are either some way of indicating that there is no message for the receiver, or returning a default message. Some message passing library, *e.g.* MPI, offers even more semantics to the receive action, permitting the receiver to request for messages from a particular sender, or of a particular type.

Blocking send/receive, and selective receive are high level semantics that are more appropriately implemented in software message passing layers or possibly NIU firmware.

2.1.7 Summary

The above review of message passing practices shows that diversity abounds. The challenge for a general purpose system is to cater to these differences without compromising the performance of any particular feature. Many of the issues covered in this review of message passing practices are inter-related, with choices in one area having implications on other aspects. For example the connectivity model (channel vs queues-and-network) has implications for buffer management, which in turn interacts with network sharing and job scheduling policy. This apparent lack of orthogonality complicates the NIU design task substantially. Section 2.3 revisits some of these issues in a multi-tasking shared network environment. Chapters 3 and 4 of this dissertation will show how our design meets these challenges.

2.2 Shared Memory Communication

This discussion focuses on coherent shared memory at the processor load/store instruction level, *i.e.* load/store instructions executed on different SMP nodes access the same logical memory location in a coherent fashion. Researchers have tried other ways of providing application programmers the abstraction of shared objects. These range from using special access routines, explicitly making a local transient copy of a shared object when access is needed [81, 50], to automated modification of executables to replace shared memory load/store instructions with access routines [95, 94]. Nevertheless, supporting shared memory at the processor load/store instruction level has better performance potential and places few requirements on software.

2.2.1 Caching and Coherence Granularity

Coherent, transparent access to shared memory using load/store instructions is commonly achieved in three ways: (i) the memory location is not cached; (ii) caching is allowed, and state information is kept at the cache-line level to maintain coherence; (iii) caching is allowed, and state information for coherence maintenance is kept at the page level.

Under the first method, each load/store instruction results in an actual remote operation. The T3D [55], T3E [96], and Tera [5] are some machines that support only this kind of coherent shared memory. Its greatest attraction is implementation simplicity; there is no cache coherence protocol to deal with. Its drawback is remote access latency on every load/store access. T3D and T3E provides special pre-fetch buffers in addition to relatively short remote access latency achieved with exotic Supercomputer class technology to reduce the effect of this latency. Tera uses special processors which exploit multi-threading parallelism to tolerate this latency.

When shared memory caching is allowed, there is the problem of a cache copy becoming stale when another processor writes over part or all of the same cache-line. The stale cached copies of the cache-line are said to have become incoherent. The most common way of maintaining coherence among caches of a bus-based SMP is through bus snooping techniques which maintains book-keeping at the cache-line granularity [7]. In distributed implementations, cache-coherence is typically maintained with a directory based approach [65, 66, 1, 2, 62] which again keeps cache-line granularity state information. Compared to uncached shared memory, caching shared memory takes advantage of data locality, both temporal and spatial, at the cost of a fair amount of book-keeping and design complexity. NIU access to system bus is needed to make this work across SMP's in a cluster.

The third approach is logically similar to the second one but does book-keeping at the page-level. This small difference translates into the big implementation advantage of requiring no hardware beyond normal paging support and some form of message passing capability. Its implementation through extensions to paging software is relatively easy in most UNIX OS's because it uses an exported interface originally meant for building third-party file systems. This approach was pioneered by Li [67], and has been further refined in many subsequent implementations, the most notable being Treadmark[53]. A variant of this approach, Cashmere [56], relies on non-coherent shared memory hardware with remote write capability.

A good shared memory system should be built from a combination of all three techniques. Either user directive, or dynamic monitoring of access characteristics can help pick the right implementation technique for a particular region of memory. Uncached access is the most efficient technique when there is little or no locality. In that case, any attempt at caching simply incurs overhead without any payback. When data is shared in a coarse grain fashion, page-level coherence is adequate and consumes little resources for book-keeping state, coherence traffic, and coherence message processing. For example, if the shared memory capability is used to simplify migration of sequential jobs for load balancing reasons, page-level coherence is perfectly adequate. Besides, most of the software infrastructure for implementing page level coherence is also needed for cluster level virtual memory paging, so supporting it incurs little additional cost.

Page-level coherence is inadequate when data is shared in a finer-grained fashion, for which cache-line granularity coherence maintenance offers better performance. As described next, cache-line granularity coherence can be implemented in several methods, each with its own advantages and short-comings. Ideally, one would like to combine the advantages while avoiding the short-comings. Our NIU is designed to facilitate this research by providing hooks for implementing the major approaches and new improvements that are being considered.

2.2.2 Memory Model

A memory model (sometimes referred to as consistency model) defines how multiple load/store operations to different memory locations appear to be ordered for code running on different processors. *Sequential Consistency* [61] is commonly regarded as the most convenient memory model for programmers. A simplified but useful way of thinking about Sequential Consistency is that in a system implementing this consistency model, all memory access operations appear to occur in the same order to all processors; it is as though all of them have been serialized. More specifically, two processors will not see a set of memory access instructions as occurring in different orders. Although this model is simple to understand, it is difficult to implement efficiently in a distributed environment.

In response to this challenge, a multitude of weaker consistency models, *e.g.* Lazy Consistency [54], Release Consistency [18], Entry Consistency [8], Scope Consistency [48], Location Consistency [34], and DAG Consistency [32], have been proposed. The main distinction between these consistency models and Sequential Consistency is they distinguish between normal memory access operations, and special order imposing operations which are typically related to synchronization. An imprecise but useful way to think about these special operations is that they demarcate points in a thread's execution where either the effects of memory operations it has executed must become visible to other threads, or those performed by other threads will be examined by this thread. By requiring the user to clearly identify these points, weaker memory models allow an implementation to achieve higher performance through delaying coherence actions and allowing greater overlap between operations. Update-based coherence protocols, for instance, can be implemented efficiently for weak memory models, but is inefficient under Sequential consistency memory model.

The question of memory model requires further research to understand the semantics and implementation implications of different models. An experiment platform, like StarT-Voyager, that is fast enough to run real programs and able to implement different memory models in comparable technology will throw much light on this issue.

2.2.3 Invalidate vs Update

A coherence protocol can either adopt an invalidate or update strategy. The former deletes cache copies that are stale, so that future accesses will incur cache-misses and fetch the new copy. The latter sends the new data to existing cache copies to keep them up to date. The two strategies are good for different data sharing patterns.

Invalidation is the best approach if a node accesses the same cache-line repeatedly without another node writing any part of it. On the other hand, update is better if there is repeated producer-consumer data communication between multiple nodes, or if there are multiple writers to the same cache-line. The latter causes thrashing in an invalidation based scheme, even if the writes are to different locations (false sharing).

Ideally, a coherence protocol should adaptively select the appropriate strategy for a cache-line (or page) based on dynamically gathered access patterns information [52]. As a nearer term goal, the user can provide directives to assist in this choice.

2.2.4 CC-NUMA and S-COMA

In the older CC-NUMA (Cache Coherent Non Uniform Memory Access) approach [65, 68], each paged-in shared memory cache-line has a unique system-wide physical address, and remotely fetched cache-lines can only reside in a traditional cache where its address tag and cache-set index furnish the global identify of a shared memory location.

Under the S-COMA (Simple Cache-only Memory Architecture) approach [39, 93], local main memory DRAM is used to cache remotely fetched cache-lines. Space in this cache is allocated at page granularity, but filled at the cache-line granularity. Because allocation is at page granularity, no address-tag at individual cache-line granularity is necessary. Instead, the function of address-tag matching in normal caches is performed by a processor's conventional virtual memory address translation mechanism when it maps virtual page number to physical page frame. Coherence state is, however, still kept at the cache-line granularity to permit finer granularity coherence control. Under S-COMA, the same logical memory location can be mapped to different local DRAM addresses on different nodes. Correspondence between each local DRAM address and a global shared memory address must be maintained as it is needed during cache miss processing.

The S-COMA approach can cheaply maintain a large cache of remotely fetched data because it incurs no address-tag overhead and uses main-memory DRAM, which is fairly cheap and plentiful, to store cache data. If, however, memory access to a page is sparse, S-COMA makes poor use of memory as a cached page of DRAM may contain only a very small number of valid cache-lines. In such cases, a CC-NUMA implementation is more efficient. Falsafi and Wood proposed an algorithm for automatically choosing between the two [29].

S-COMA and CC-NUMA are two ends of a spectrum in which middle-ground schemes are possible. By introducing a small number of address-tag bits for each cache-line in S-COMA DRAM, each S-COMA DRAM page can be shared between several physical page frames. At any one time, each DRAM cache-line can only be used by a particular page frame, but cache-lines from several physical page frames can use different cache-lines of a DRAM page.

The number of address-tag bits is kept small by requiring the physical page frames mapped to the same S-COMA DRAM page to have page frame addresses that differ only in a small number of bits in a pre-determined bit field. In addition, the memory controller has to alias these pages to the same DRAM page by ignoring this bit field. For example, a system may allow up to four physical page frames to map to the same DRAM page with the constrain that these page frames must have the same addresses except in the most significant two bits. In this case, only two bits of address-tag is kept for each DRAM cache-line to identify the actual physical page frame using it.

This scheme, which we call *Nu-COMA* (Non-Uniform COMA), combines S-COMA's benefit of full-associative mapping of virtual pages to physical page frame, with the ability to share a DRAM page between several virtual pages. Furthermore, the address-tag overhead is small, unlike the case of a traditional cache. Another interesting feature of Nu-COMA is when a single DRAM cache-line is shared between two active (physical address) cache-lines. Nu-COMA does not required switching the DRAM cache-line between the two cache-lines which can lead to thrashing[1]. Instead, one cache-line is allowed to use the DRAM in the traditional S-COMA way while the

[1]Caches closer to the processor may or may not suffer from thrashing depending on details of their organization, such as associativity and number of sets in the caches.

other is treated like a CC-NUMA cache-line.

From an implementation perspective, CC-NUMA can be implemented with a subset of the infrastructure needed for S-COMA. Nu-COMA will require additional, simple hardware support beyond those used for S-COMA and CC-NUMA.

2.2.5 Atomic Access and Operation-aware Protocol

A shared memory implementation must include atomic access support. In systems implementing weak memory models, synchronization actions, including atomic accesses, must be dealt with specially because they take on additional semantics for coherence maintenance. Furthermore, coherence protocol for locations used as synchronization variables should be different because they are often highly contended for, and are accessed in a highly stylized manner. As an example, QOLB (Queue on Lock Bit) [37, 51] is proposed as a means of avoiding unnecessary cache-miss fetches for locations used as mutex locks. One can regard it as an instance of a class of *operation-aware* protocols which are cognizant of the semantics of and operations on data values stored in the shared memory locations.

Operation-aware protocols may use local copies of a memory location to maintain parts of a distributed data structure instead of simply being a copy of a variable. Furthermore, the "coherence operations" perform functions other than simply updating or invalidating a cache-copy. For instance, in the case of a lock, a special lock protocol may ensure that at most one local copy indicates the lock to be free (say a non-zero value) while other local copies all indicate that the lock is not free (say a zero value). "Coherence maintenance" on this location includes the responsibility for moving the "lock is free" instance from one node to another across the cluster as the need arises, and maintaining a list of waiting lock requesters.

Aside from locks, memory locations used for accumulating the sum, minimum, maximum, or some other reduction operation can also benefit from operation-aware coherence protocols. The protocol could automatically clone local versions of the memory location, and combine the local version with the global version periodically. A special operation is then used to obtain the globally up-to-date value. Further research and experimentation is needed to evaluate the effectiveness of operation-aware protocols, but as a start, the NIU must be able to accommodate their implementation.

2.2.6 Performance Enhancement Hints

Shared memory performance can often be improved with hints. A system should allow easy specification of hints such as the choice of coherence protocol, pre-fetching and pre-sending of data of various granularities. Because hints only affect performance but not correctness they do not have to be precisely accurate all the time.

The NIU hardware should also assist in collecting memory access statistics. Either user code or the cache coherence protocol can then use this information to select the appropriate coherence maintenance strategy.

2.2.7 Summary

Overall, because shared memory support is still an active research area, it is important that our NIU design permits experimentation. Both CC-NUMA and S-COMA styles of implementation should be supported. The implementation should permit easy modifications to coherence protocol and provide low-cost mechanisms for dynamically relaying hints from the application code to the underlying CCDSM implementation.

2.3 System Requirements

System-level requirements for cluster communication center around protection enforcement and fault isolation. Protection enforcement depends on the network sharing model, which is in turn highly related to the job scheduling model. Fault isolation focuses on preventing faults at one SMP from spreading to other SMP's through cluster level communication.

2.3.1 Multi-tasking Model

Existing approaches to job scheduling for parallel applications have mostly been restricted to gang scheduling. When there is tight dependence between different threads of execution, gang scheduling is necessary to ensure that threads scheduled to run are not held back waiting for data from unscheduled threads. This led to the popularity of gang scheduling and the side-effect that existing protection schemes in message passing systems are mostly built on this assumption.

Unfortunately, gang scheduling is too restrictive for a general cluster system. For one, it is not an appropriate concept for distributed application and OS communication. OS communication is more naturally thought of as occurring on-demand, concurrently with user jobs' communication. If the OS message is urgent, it should cause an interrupt at the destination so that it is handled immediately. If it is not urgent, the message should be buffered up for subsequent processing, such as when the OS takes an exception to service some other trap or interrupt.

It is also unclear at this point how a job scheduler should deal with exceptional events, such as page faults, on a subset of a parallel job's processes. Many earlier parallel machines have side-stepped this issue by operating with no virtual memory (*e.g.* CM-5, most Crays). Suspending the entire parallel job when exception occurs runs the danger of causing repeated job swaps as page faults occur in one process after another of the same parallel job. A better strategy is to allow the unaffected processes to continue until data dependence prevents them from making further progress.

More generally, parallelism variation across different phases of a program's execution suggests that machine utilization will improve if the number of processors devoted to a parallel job can expand and contract over time. A parallel job may have processes that are fairly dormant during certain phases of execution. At such times, the under-utilized processors should be allocated to other jobs, including other parallel jobs or subset of a parallel job. A parallel job with coarse granularity dependence, for example, is a good candidate for absorbing the idle processors as it is likely to make useful progress without all its constituent parts scheduled in a gang fashion.

The cluster communication support can make it easier to improve job scheduling by imposing fewer restrictions on the scheduler. Thus, the traditional practice of using gang scheduling to impose communication protection by limiting the number of jobs simultaneously sharing the network are unnecessarily limiting. The communication support should also be virtualized so that it does not impose any impediment on process migration. If a job scheduler can easily and cheaply migrate processes, it will have an easier task at load balancing. The same capabilities also make it easier to implement automatic check-point and restart. Along the same line, a communication architecture with appropriate hooks can ease the implementation of run-time expansion and contraction of allocated processor resource.

2.3.2 Network Sharing Model

New network sharing and protection models are needed to achieve greater job scheduling flexibility. Our discussion will focus on network sharing in the presence of direct user-level network access. With the exception of experimental machines like Alewife and possibly FLASH (depending on the PP firmware in MAGIC), the network in distributed shared memory machines is not directly accessible from user-level code and is really a private resource controlled by the coherence protocol. As a result, there is no issue of isolating network traffic belonging to different protection domains. Job scheduling on these machines is not restricted by any communication protection concerns, which is adequately taken care of by normal virtual address translation. The picture changes immediately when user-level code is given direct access to the network. We first examine how a number of message passing machines deal with sharing and protection issues, before proposing our own model.

The CM-5 treats its fast networks as a resource dedicated to each job while it is running, much like processors. The machine can be shared by several independent jobs in two ways: (i) it can be physically partitioned into several smaller independent units; and (ii) each partition can be time-sliced between several jobs under a strict gang scheduled policy. Within each partition and time-slice, its networks are not shared. Context switching between time slices includes saving and restoring the network state with the assistance of special operation modes in the CM-5 network switches. Aside from having a rather restricted network sharing model, this design employs special context-switching features in switch hardware which are not useful for other purposes. Context switching a lossless network without this level of support is tricky as one runs into the issue of buffer space requirement for the packets that are swapped out. Unless some flow-control scheme is in place to bound packet count, the storage requirement can build up each time a job is swapped in and out. Furthermore, unless care is taken, a save-and-restore scheme is also likely to affect delivery order guarantees.

The SP-2 adopts a slightly different solution, but essentially still gang schedules parallel jobs and uses the network for only one user job. Instead of having hardware capability for context-switching the network, the SP-2 NIU tags every out-going message with a job ID. At the destination, this ID is checked against that of the current job. If a mis-match is detected, the message is dropped. The scheme relies on a message loss recovery protocol to re-send the dropped message at some later time. This allows each context switch to be less strictly synchronized. SP-2 also allows system code to share the network by supporting an extra system job ID and message queues.

We argue for sharing the network aggressively. As one of the most expensive resource in a cluster system, the network should be used by all traffic that benefits from fast communication. In addition to user parallel jobs, the fast network should be available to OS services such as a parallel file system and the job scheduler. To support the flexible job scheduling policies described earlier, the network must be prepared to transport messages belonging to arbitrarily many communication domains.

To protect the independence of two parallel jobs A and B that happen to be sharing a network, the communication system must ensure that job A cannot intercept job B's messages nor fake messages to job B, and vice-versa. Furthermore, a job must be prevented from hogging shared network resources and depriving other jobs of communication services.

The SP-2 job ID solution can be generalized to satisfy these requirements. But traditional loss recovery protocol often increases communication latency because of more complex source buffering requirements and protocol book-keeping. This is particularly troublesome with the queues-and-network model, because sliding window protocol is a per source-destination pair protocol. This mismatch means that a simple FIFO send buffer management policy, which is convenient for hardware implementation, does not work. Instead, when retransmission is needed, send buffers may be read and deallocated in an order different from the message enqueue order. The increased hardware complexity may increase latency beyond what is acceptable for latency sensitive communication such as shared memory protocol. A solution that permits simpler hardware, at least in the common case, is highly desired.

A shared network also interacts with buffer management in that it cannot afford to allow packets to block into the network when destination queues are full. This option, allowed in the CM-5 and most distributed shared memory machines, requires logically separate networks. While these logically separate networks can be implemented as virtual channels in the network switches to avoid the cost of extra wires and package pins, a fast network switch cannot support more than a small number of virtual channels. The virtual channel solution is therefore insufficient for our environment, where an arbitrary and large number of communication domains have to be supported simultaneously.

Issues of network sharing are present in LAN and WAN, but critical differences between these and tightly coupled cluster networks make many of the LAN/WAN solutions infeasible. Communication occurs much more frequently in cluster system and requires much lower latencies. Because these differences usually span several orders-of-magnitude, LAN/WAN protection approach of implementing communication through system calls was long ago recognized as too expensive. For the same reasons, we are re-visiting the practice of using common loss recovery protocols, such as variants of sliding window protocols, for preventing network blockage. New solutions could exploit positive attributes of cluster system networks such as its extremely low communication latency.

2.3.3 Fault Isolation

In order for a cluster system to be more reliable than its constituent SMP's, faults in each SMP must be isolated. Each SMP should run its own instance of the operating system that can function in isolation. Parallel or cluster wide services should be layered on top as extensions. By limiting the interaction of OS's on different SMP nodes to a well defined, protected interface, it is easier to isolate faults.

At a lower level, it is necessary to impose memory access control on remotely initiated shared memory operations, particularly remotely initiated memory writes. Granting each remote SMP access to only a limited part of memory confines damage that each SMP can wrought. Local OS's state, for example, should not be directly read/write-able with shared memory operations from any other SMP.

2.4 SMP Host System Restrictions

A typical commercial SMP offers an NIU limited points of interface. The most common options are the I/O bus and the memory bus; interfacing via DRAM SIMM/DIMM interface has also been proposed.

The I/O bus is by far the most popular choice. Examples include the SP-1 and SP-2 [3], Myrinet [11], StarT-Jr [43], StarT-X [42], Memory Channel [36], and Shrimp [10][2]. The I/O bus, designed to accommodate third party devices, has the advantage of being precisely specified. There are also extensive tools, chip-sets and ASIC modules that are available to make implementation easier. Unfortunately, current bridges connecting I/O and memory buses do not propagate sufficient cache operation information for efficient cache-coherent distributed shared memory implementation.

Interfacing at the main memory DRAM interface was proposed in MINI [38]. Just like in the case of I/O bus, insufficient cache-state information is exchanged across the memory bus/DRAM interface for implementing cache-coherent distributed shared

[2]Shrimp-II has interfaces on both the EISA I/O bus and the memory bus.

memory efficiently. Its advantage over an I/O bus NIU is potentially shorter latency for message passing operations, due to its proximity to the processor.

The memory bus is the most flexible point of interface for a cluster system NIU. An NIU positioned on the memory bus can participate in the snoopy bus protocol, making cluster-wide cache-coherent shared memory implementation feasible. Message passing performance is also good due to close proximity to both the SMP processors and memory, the two main sources and destinations of message data. Message passing interfaces can also take advantage of the coherent caching capability of the memory bus. The NIU of most existing shared memory machine are located here, *e.g.*, DASH [66], Origin-2000 [62], NUMA-Q [68], SynfinityNuma [102]. Message passing machines like the CM-5, the CS-2 [44], and the Paragon [49] also have NIU's on the memory bus. This choice is convenient with today's SMP's, whose multiple memory bus slots, intended for processor, and sometimes I/O cards, present ready slots for the NIU.

This thesis will investigate an NIU that interfaces to the SMP memory bus. Although the design of such an NIU is influenced by the types of bus transaction supported by the SMP, these are fairly similar across today's microprocessors so that the general architectural principles proposed in this work are application across SMP families. As an example of the uniformity among SMP families today, practically all of them support invalidation-based cache-coherence. Most will also allow additional system bus devices to be bus master, slave, and snooper. The main variations, presented below, are the atomic access primitive, and the method of intervention. These differences affect low level NIU design choices, but have no bearing on the high level architectural concepts of our thesis.

The most common atomic access operations are the load-with-reservation (LR) and store-conditional (SC) pair used in PowerPC, Alpha and MIPS. Swapping between a register and a memory location is used in Sparc and x86 processor architectures. Sparc also supports conditional swap. There is as yet no definitive position on how each of these atomic operations scale in a distributed system.

Intervention refers to the process whereby a cache with the most up-to-date copy of a cache-line supplies the data to another cache. Less aggressive snoopy buses implement intervention by retrying the reader, having the cache with the data write it back to main memory, and then supplying data to the reader from main memory. More aggressive buses simply have the cache with the up-to-date data supply it directly to the reader, possibly with write-back to main memory at the same time. This is sometimes called *cache-to-cache transfer*. Many systems which implement the less aggressive style of intervention can accommodate a look-aside cache. This supplies data directly to readers, providing a special case of cache-to-cache transfer. We will see later that our NIU design makes use of this feature.

Interfacing to the memory bus has the slight disadvantage of dealing with a proprietary, and sometimes not very well documented bus. This is not a serious problem if the SMP manufacturer is accessible to clarify ambiguities. For portability of network interfaces, one may choose to define a new interface designed to accommodate cluster communication, but this is beyond the scope of this thesis.

2.5 NIU Functionalities

This section examines the tasks an NIU performs to deliver the communication functions and system environment described in earlier sections of this chapter. We break communication operations into elementary steps that are examined individually. We begin with three key functions supplied by every NIU: interface to host, interface to network, and data path and buffering. Next, two other functions commonly found in NIU's are discussed; these are: data transport reliability and ordering, and support for unilateral remote communication action. Finally, cache-coherent shared memory related NIU functions, partitioned between a cache protocol engine, and a home protocol engine, are described.

2.5.1 Interface to Host

As the intermediary between a computation node and the network, an NIU is responsible for detecting when communication is required, and determining the communication content. It is also responsible for notifying the host node of information that has arrived and making that information available.

Except for NIU's that are integrated into microprocessors, an "extinct species" these days, NIU's typically detect service requests by the following methods: (i) monitoring the state of pre-determined main memory locations, (ii) presenting software with a memory mapped interface, or (iii) snooping bus transactions. The first two methods are commonly employed when communication is explicitly initiated by software, as in the case of message passing style communication. Among the two, method (i) has the disadvantage that the NIU has to actively poll for information. Method (ii) allows event-driven processing in the NIU. The last approach is most commonly used to implement shared memory, where communication is implicitly initiated when software accesses to shared memory locations trigger remote cache-miss or ownership acquisition processing. The method can also be used in combination with (i) to reduce polling cost: the pre-determined location is only polled when snooping indicates that a write has been attempted on that location.

Message passing communication requires the NIU to inform the host when information arrives for it. This may be done actively, with the NIU interrupting a host processor, or passively with the NIU supplying status information in either predefined main memory locations, or memory mapped NIU registers. In the case of shared memory, software on the host processor never explicitly sees the arrival of messages. Instead, incoming information is delivered at the hardware level; *e.g.*, replies to locally initiated requests, such as cache-miss or cache-line ownership, allow pending bus transactions to complete. Externally initiated requests, such as to remove cache copies or write permission of a cache-line from local caches, result in NIU initiated system bus transactions that cause the desired changes.

The different methods of detecting service requests and delivering message availability information require different NIU capabilities. If coordination between NIU and host is through the main memory, the NIU needs to have bus master capability, *i.e.* be able to initiate bus transactions. If coordination is through memory mapped NIU registers, the NIU has to be a bus slave, a device which responds to bus transactions, supplying or accepting data. As a snooper, the NIU has to participate in snoopy bus protocol, intervening where necessary. Although limiting the NIU to be either just a slave, or just a master simplifies the NIU design, an NIU with all three capabilities, master, slave and snooper, has the greatest flexibility in host interface design.

2.5.2 Interface to Network

The NIU is the entity that connects directly to the network. As such, it has to participate in the link-level protocol of the network, behaving much like a port in a network switch. Link-level protocol covers issues like how the beginning and end of a packet is indicated, and flow-control strategy to deal with possible buffer over-run problems. In some networks [25, 33], it also includes link-level error recovery. This part of the NIU also deals with signal encoding (*e.g.* Manchester encoding to improve reliability), and the actual electrical levels used in signaling.

Data transported over the network has to conform to some format specified by the network. For instance, there is usually a header, containing pre-specified control fields like the packet destination, followed by payload of arbitrary data. Most systems also append extra information for checking the integrity of the packet, usually some form of CRC. A network may also impose a limit on the maximum packet size; messages that are larger than the size limit will have to be fragmented at the source and reassembled at the destination. Whether these details are directly exposed to software, making it responsible for assembling messages into the network packet format, or masked by the NIU vary across systems.

2.5.3 Data Path and Buffering

An NIU provides the data path between the network and its host node. This includes at least some rate-matching buffers used as transient storage to decouple the real-time bandwidth requirement of the network from that available on the host node.

Within the node, CC-NUMA style shared memory directly transfers data between the NIU and caches. Although S-COMA style shared memory logically moves data between the NIU and main memory DRAM, actual data transfer sometimes occurs directly between the NIU and caches.

Message passing often involves additional buffers which can be thought of as service delay buffers. For instance, incoming messages can be buffered away until software is at a convenient point to service them. Such buffers are managed in a co-operative fashion between node software and the NIU.

Service delay buffers may be on the NIU or in main memory DRAM. Another possibility is to place them logically in the main memory, but have the NIU maintain a cache of them. For NIU's located on memory buses that support cache-to-cache intervention, the latter design avoids cycling data through DRAM in good cases, while having access to large, cheap buffer space in main memory.

Every NIU has to deliver all the above three functions in some form. In fact, simple message passing NIU's provide only these functions and none of those described later. For instance, the CM-5 NIU is a slave device on the memory bus, providing memory mapped FIFO's that software on the node processor writes to or reads from directly. These FIFO's serve both the bandwidth-matching and service delay functions.

The StarT-X PCI bus NIU card [42] is both a slave and a master on the PCI bus. As a slave, it accepts direct command and data writes from the processor, much like the CM-5 NIU, except that the writes physically passes through a chip bridging the memory and PCI buses. With its bus master capability, the StarT-X NIU can transfer data to and from main memory DRAM. This allows it to use main memory DRAM as service delay buffers.

2.5.4 Data Transport Reliability and Ordering

In many systems, the function of recovering from losses caused by unreliable network service, or NIU dropping packets is left to software, *e.g.* most ATM networks.

Although this does not present a functional problem since software can implement recovery protocols, it is best that the NIU provides this function in low-latency networks. Aside from off-loading this overhead from software, the NIU is in a better position to implement the recovery protocol – without the duty of running long computation threads, it can send and handle acknowledgements in a timely fashion and monitor the need for re-transmission. For similar reasons, protocols for guaranteeing in-order communication are also best implemented by the NIU.

NIU implemented loss recovery and ordering protocols is most often found in message passing NIU's with programmable embedded processors where the function is implemented in firmware. Examples include the SP2, and the Myrinet. Most shared memory machines relied on their networks to guarantee reliable data transport. This is strongly motivated by their requirement for low communication latency. An exception is the SynfinityNuma [102] which has hardware implemented NIU-to-NIU loss recovery protocol.

2.5.5 Unilateral Remote Action

A few NIU's, such as Shrimp [10], Memory Channel [36], T3D [55] and T3E [96], support uncached remote memory access. This ability to carry out remote actions unilaterally can be a very useful because it decouples the round-trip communication latency from the scheduling status of any user software counterpart on the remote end. Currently, this capability is only available for stylized operations, typically memory read/write. The T3D also support an atomic remote fetch-increment-write operation. The ability to run short remote user threads upon message arrival has been proposed, *e.g.* hardware platforms like J-machine [24] and M-machine [30], and sofware libraries like Active Message [100], but has not found wide-spread adoption.

2.5.6 Cache Protocol Engine

The NIU of a cache-coherent distributed shared memory system performs tasks beyond those listed above, namely coherence maintenance. Though traditionally considered part of the cache controller or the memory controller, coherence functions are included in our NIU because they are a form of communication and many capabilities needed to implement shared memory are also useful for message passing. Coherence hardware is divided into two parts: cache protocol engine described in this section, and home protocol engine described in the next section. Our discussion is confined to NIU's with direct access to the system bus because other points of interface are inadequate to this task.

The cache protocol engine performs the cache side of coherence shared memory support. It has two functions: (i) it determines when a requested operation on a cache-copy requires actions at other nodes and initiates those actions; and (ii) it executes other nodes' commands to modify the state or data of local cache copies. The cache protocol engine is really part of the NIU's interface to host.

For S-COMA style shared memory implementation, the cache protocol engine has to maintain cache-line state information, and implement access control to DRAM cache-lines. For CC-NUMA style shared memory, the cache protocol engine typically does not retain any long term information, but simply "echoes" bus transactions to the home site for the cache-line involved.

Executing other nodes' commands on local cache copies involves performing an appropriate system bus transaction. This is of course limited to the set of bus transactions supported by the host system. In the case of S-COMA, the cache protocol engine may also have to modify the cache-line state it maintains.

An important role of the cache protocol engine is to maintain transient state for out-standing operations. Because a cache-coherence protocol is a distributed algorithm dealing with concurrent activities, a local view sometimes cannot differentiate between a number of possible global scenarios. The projections of these scenarios onto a local node can be similar, but the scenarios require different local actions. Maintaining transient state can help disambiguate. For instance, when a cache protocol engine receives an external command to invalidate a cache-line for which it has a pending read, it is not possible to tell whether the command has over-taken a reply to its pending read. (Many implementations put the two messages onto different virtual networks due to deadlock concerns; thus message ordering cannot be assumed.) The cache protocol engine therefore has to be more sophisticated than echoing commands/requests in a simplistic way.

Support for S-COMA requires the NIU to track address mapping, so that from the physical address of a cache-line, the cache engine can determine its home site and provide the home site with sufficient information to look up the cache-line's directory information. The cache engine also needs to track global address to local physical address mapping so that home protocol engines operate on globally addresses only.

CC-NUMA style shared memory avoids address translation by using a fixed field of the physical address to identify the home node. As long as a large physical address space is available, this is an adequate approach. Translation capability bring convenience, *e.g.* when the home node is migrated, only the translation information needs to be modified instead of having to flush all existing cache copies.

2.5.7 Home Protocol Engine

Associated with main memory, the home protocol engine is the centralized controller which maintains the global view of what is happening to a cache-line. It is typically the authority which issues caches the permissions to maintain and operate on cache copies. It also initiates commands to these caches to modify those permissions. The exact functions performed by the home protocol engine differ across implementations. For instance, almost all systems keep directory for each cache-line to identify the caches that (may) have copies of that cache-line. This can be kept in a localized data structure at the cache-line's home site (the common approach), or in a distributed data structure, maintained co-operatively with the cache engines, as in SCI [47].

Directory size is often a small, but significant fraction of the supported shared memory size. While highly implementation dependent, this fraction is often targeted to be not more than 10 to 15%. For example, if 8 bytes is kept for every 64 bytes cache-line, the overhead is about 12%. It is also preferably to use part of main memory DRAM for this purpose, but when that is not feasible (such as due to possible deadlocks), extra DRAM can be provided on the NIU.

Earlier shared memory machines with only CC-NUMA support include DASH, Alewife[3], SGI's Origin-2000, Sequent's NUMA-Q (subsequently renamed STiNG), and Fujitsu/HAL's SynfinityNuma.

[3]Alewife's coherence protocol is implemented with a combination of hardware FSM's and processor software. Hardware handles the common cases but interrupts the processor for corner cases. Therefore, it is theoretically possible for Alewife to implement S-COMA style shared memory. This has not been attempted.

Chapter 3

A Layered Network Interface Macro-architecture

This chapter presents our NIU's macro-architecture. The macro-architecture is an abstract functional specification realizable with multiple implementations, one of which is presented in Chapter 4. We adopt a layered architecture to facilitate implementing a wide range of communication operations and ease of porting. Three layers are defined: (i) a Physical Network layer, which provides reliable, ordered packet delivery; (ii) a Virtual Queues layer, which implements protected network sharing; and (iii) an Application Interface layer, which implements the communication interface exported to application code, for example shared memory or message passing operations.

Each layer builds on services provided by the layer below to offer additional functions. By dividing the responsibility this way, a uniform framework takes care of network sharing protection issues once and for all, and the Application Interface layer can implement multiple abstractions without worrying about subtle protection violation; it only needs to control, through traditional virtual memory translation mechanisms, which job gets access to each memory-mapped interface.

Figure 3-1 compares our proposed communication architecture against that of more established machines. Each NIU in the other machines has a monolithic structure, and presents a small set of interface functions. The figure also shows how each architecture has a different way of enforcing sharing protection. Protection in our

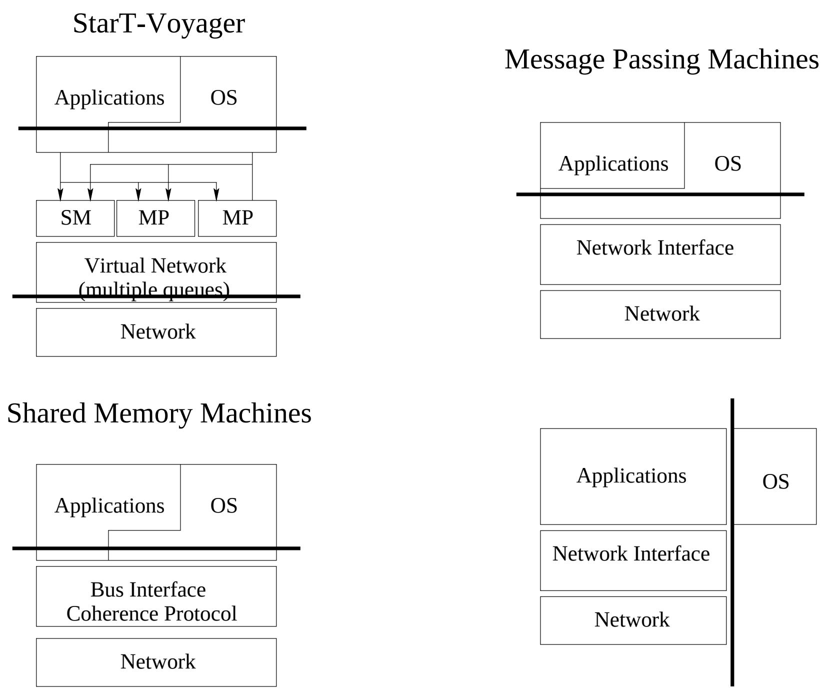


Figure 3-1: Comparison of StarT-Voyager network interface architecture against those of traditional message passing machines and shared memory machines. The bold black lines indicate points where protection checks are imposed. Message passing machines typically either rely on OS to enforce communication protection (top right), or operate in single network user mode (bottom right).

architecture, demonstrated in the StarT-Voyager example, is enforced in two different places. The first one, higher up in the diagram, makes use of conventional VMM translation, while the second one is implemented in the Virtual Queues layer in the NIU.

3.1 Physical Network Layer

The Physical Network layer defines the characteristics of the underlying data transport service. These are chosen both to ease the design and implementation of the layers above, and to match the functions supplied by most system area networks (SAN). The Physical Network layer provides reliable, ordered packet delivery over two logically independent networks, with bounded number of outstanding packets in each network.

3.1.1 Two Independent Networks

Abstractly, the Physical Network layer at each NIU provides two pairs of send and receive *packet* queues corresponding to entry and exit points of two independent networks. Each packet enqueued into a send queue specifies its destination receive queue, which has to be on the same network. Having more than one network greatly increases the options for over-coming network deadlock problems, with attendant performance improvements.

3.1.2 Reliable Delivery Option

The Physical Network layer provides reliable packet delivery as an option on a *packet by packet basis*. Packets that are not marked as requiring reliable delivery may never reach its destination queue, but are never delivered more than once. This service guarantee is made an option because we expect some systems to incur lower overhead for packets that do not request for reliable delivery.

An alternative is to push delivery reliability guarantees up to the next layer, the Virtual Queues layer. Our discussion of buffer management in Section 2.1 mentioned that a design which dynamically allocates receiver queue buffer space at message arrival may run into the situation where no buffer space is left. If not handled properly, that problem can lead to deadlock affecting all parties sharing the network.

We will revisit this problem in Section 3.2.2. For now, it suffices to mention that one solution is to drop a packet when its destination queue is out of buffer space, and recover with a loss recovery protocol implemented between the private messages queues of the Virtual Queues layer. Attempting to recover the dropped packet with loss recovery protocol at the shared queues of the Physical Network layer does not resolve the deadlock (more details in the next section). It therefore appears that requiring reliable packet delivery at the network level is redundant.

Our choice is motivated by two performance considerations. Firstly, many cluster networks provide reliable packet delivery service which one would like to pass on to application code. This level of service is wasted if network sharing entails packet dropping and the attendant overhead of loss recovery protocol.

Secondly, implementing a sliding window protocol requires a fair amount of state and buffering. If it is done at the multiple queues level, it most likely will have to be done in NIU firmware. Implementing it directly in hardware for all the queues in the Virtual Queues layer is infeasible because of the significant increase in per message queue state. In contrast, by pushing the loss recovery protocol down to the shared Physical Network layer, each NIU only implements two copies of the protocol, one for each network, making it feasible to implement the protocol fully in hardware at high performance.

Finally, our choice is possible because of an alternate solution to the deadlock problem: Reactive Flow-control. This is discussed in Section 3.2.3.

3.1.3 Ordered Delivery Option and Ordering-set Concept

We introduced a more flexible concept of delivery ordering to permit taking advantage of multiple paths in a network. Each packet which requires ordered delivery of some form specifies an *ordering set*. Packets between the same source-destination pair that have requested for ordered delivery under the same ordering-set are guaranteed to be delivered in the sent order. No ordering is guaranteed between packets of different ordering-sets. The traditional notion of message ordering is the limiting case where only one ordering-set is supported.

The ordering-set concept is useful in networks which provide multiple paths between each source-destination NIU pair. For systems that determine packet routes at the source, the ordering-set concept translates into using the same path for packets in the same ordering set. Ordered packets from different ordering-set can utilize different paths, spreading out traffic load. Packets with no ordering requirements are randomly assigned one of the possible routes.

3.1.4 Bounded Outstanding Packet Count

The term *outstanding packets* refers to packets in transit in the network: they have left the source NIU, but may not have reached the destination NIU. A bound on the number of outstanding packets is required by the Reactive Flow-control scheme described in Section 3.2.3. A well chosen bound can also prevent congestion in the network, without limiting the good case network performance.

Packet delivery reliability was made optional based on the argument that providing that guarantee incurs additional cost in certain system. Given the need to account for outstanding packets, that argument may appear invalid. After all, accounting for outstanding packets requires knowing when a packet no longer exists in the network. If one has gone through the effort of tracking the existence of a packet, recovering from its loss should not be that much more expensive.

This perception is not strictly true. Packets may be lost because they are deliberately dropped at the destination NIU in response to lack of buffer space in destination queues. Accounting for outstanding packets is relatively simple, with destinations sending periodic, aggregated acknowledgement counts to the sources. In comparison, if recovery of lost packets is needed, it becomes necessary to either buffer packets at the source until delivery is confirmed. A bound on the number of outstanding packets can also be gotten in passive ways, *e.g.*, the entire networks' transient buffering capacity provides a bound which may be low enough to be useful for Reactive Flow-control.

3.1.5 Programmable Send-rate Limiter

A programmable send-rate limiter imposes a programmable minimum time interval between launching two packets into the same network. It is a useful tool for congestion control. Congestion in the network arises when senders inject packets at a faster rate than they can be removed. In switched networks, this often leads to very bad congestion due to tree-blocking effects. With the send-rate limiter, faster NIU's in heterogenous environment can be programmed to match the receive speed of slower

NIU's [45]. The minimum time interval can also be dynamically altered in response to network conditions.

The Physical Network layer implements the glue to the underlying network. When that network already possesses the desired properties, the network layer simply implements low level signaling between the NIU and the network. When there is a mismatch, some form of sliding window protocol can be used to bridge the gap. Sliding window protocol not only recovers from packet losses, but can also enforce order. Furthermore the send window size imposes a bound on the number of (unique) outstanding packets.

3.2 Virtual Queues Layer

The Physical Network layer is a physically addressed packet transport service, unaware of processes and protection requirements. The Virtual Queues layer virtualizes it, laying the foundation for multi-tasking sharing of the network. System-level job scheduling flexibility depends on this layer. Our design not only facilitates transparent process migration, but also enables novel features such as dynamic contraction and re-expansion of the number of processes devoted to a parallel job.

Figure 3-2 shows the Virtual Queues layer and Physical Network layer working co-operatively. Central to the Virtual Queues layer is the abstraction of an *arbitrary number of message queues*, allocated by the system as private queues to different jobs. These queues form the components for constructing an *arbitrary number of independent virtual networks* which can operate simultaneously. Protection is strictly enforced through a virtual queue naming and translation scheme, while the out-of-receive-buffer problem is solved by a Reactive flow-control protocol which imposes zero performance cost under favorable conditions.

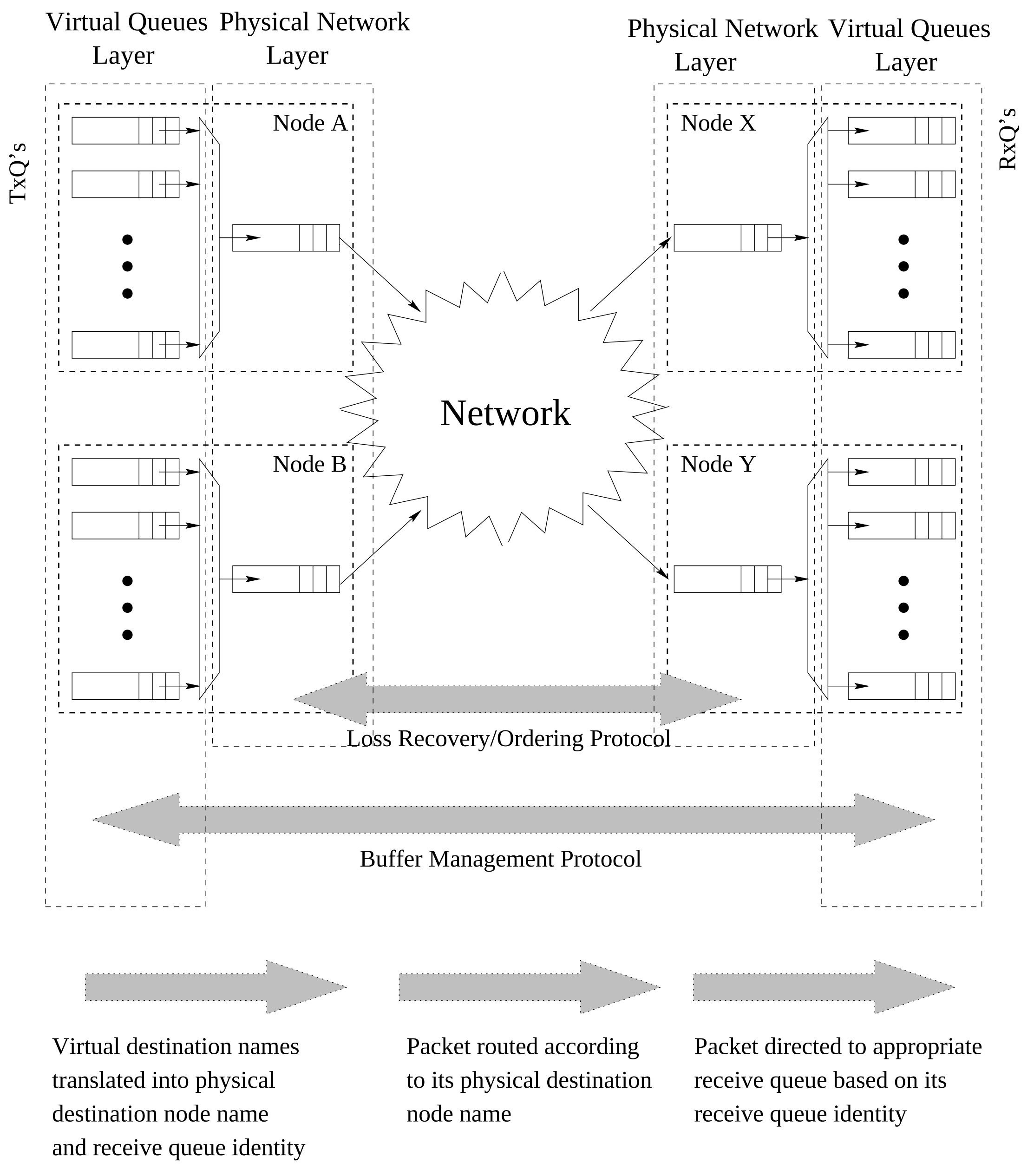


Figure 3-2: Relationship between the Virtual Queues layer and Physical Network layer.

3.2.1 Virtual Queue Names and Translation

Controlling access to message queue is the basis for protection. There are two categories of message queue access: (i) local access to transmit and receive queues for message enqueue and dequeue operations respectively; and (ii) remote receive queue access, *i.e.* naming a receive queue as the destination of a message.

Without modifications to microprocessors, local access to transmit and receive queues has to be through a memory mapped interface. Access control here is then simply a matter of restricting which transmit and receive queues are mapped into the virtual address space of a process. The normal virtual address translation mechanism takes care of enforcing protection in each access instance.

Control over the naming of message destination queue is more interesting as it defines the communication domains. Application code refers to message destinations with virtual names, each of which is translated at the sender's NIU into a *global queue address* used by the Physical Network layer. This translation is context dependent, with each transmit queue having a different destination name space. Only system code can set up the translation tables, which define the "reach" of each transmit queue.

Each global queue name has two components: one part identifies the destination NIU, while the second specifies a receive queue identifier (RQID) on that NIU. The latter is subjected to a second translation at the destination NIU to determine the actual destination queue resources. The utility of this second translation will become apparent later when we discuss process migration in Section 3.2.5 and the combined Resident and Non-resident queues approach of implementing the Virtual Queues layer in Section 4.6.2.

This scheme allows arbitrary communication graphs to be set up. Aside from the fully-connected communication graph common to parallel jobs, partially connected communication graphs are also possible. The latter may be useful for client-server communication where a server is connected to its clients but the clients are not connected to one another. Adding and removing connections dynamically is also easy in this scheme.

Our protection scheme for network sharing offers several advantages over conventional approaches, such as network time-slicing employed on the CM-5, and the job-ID tagging and matching scheme used on the SP-2. These approaches coupled three issues which we feel should be dealt with separately; these are: (i) communication protection, (ii) job scheduling; and (iii) network packet loss recovery. Coupling communication protection with jobs scheduling restricts the kind of job scheduling or network sharing in a system. As for network packet loss recovery, many cluster system networks are reliable enough to not require provision for recovery from losses due to network errors.

Our protection scheme also offers other system-level benefits like easing job migration and dynamic computation resource adjustment as described below. But before that, we show that our scheme can be viewed as a generalized job-ID tagging scheme.

The job-ID tagging scheme on SP-2 only delivers packets that are tagged with either the current job's job-ID, or the system job's job-ID. It is easy to extend this scheme so that the receiver NIU permits more receive queues to be active, and demultiplexes incoming packets into their desired receive queues. This is, however, insufficient when we want to migrate jobs and their message queues that used to be on different nodes to the same node. Job-ID alone does not provide sufficient disambiguation; it has to be further refined by some kind of process identity or queue identity. The is essentially our destination queue naming and translation scheme.

3.2.2 Dynamic Destination Buffer Allocation

This section revisits the problem of allocating destination queue buffer space dynamically at message arrival time. When space is unavailable, the choices are either to drop or block the message. Blocking is a dangerous option in a shared environment because a blocked packet occupies a shared network buffer leading to the insidious build up of dependences *between* logically unrelated jobs. Blocking is only safe if there is some way to guarantee that space will eventually free up. In dedicated network environments, this typically depends on the application respecting some network usage

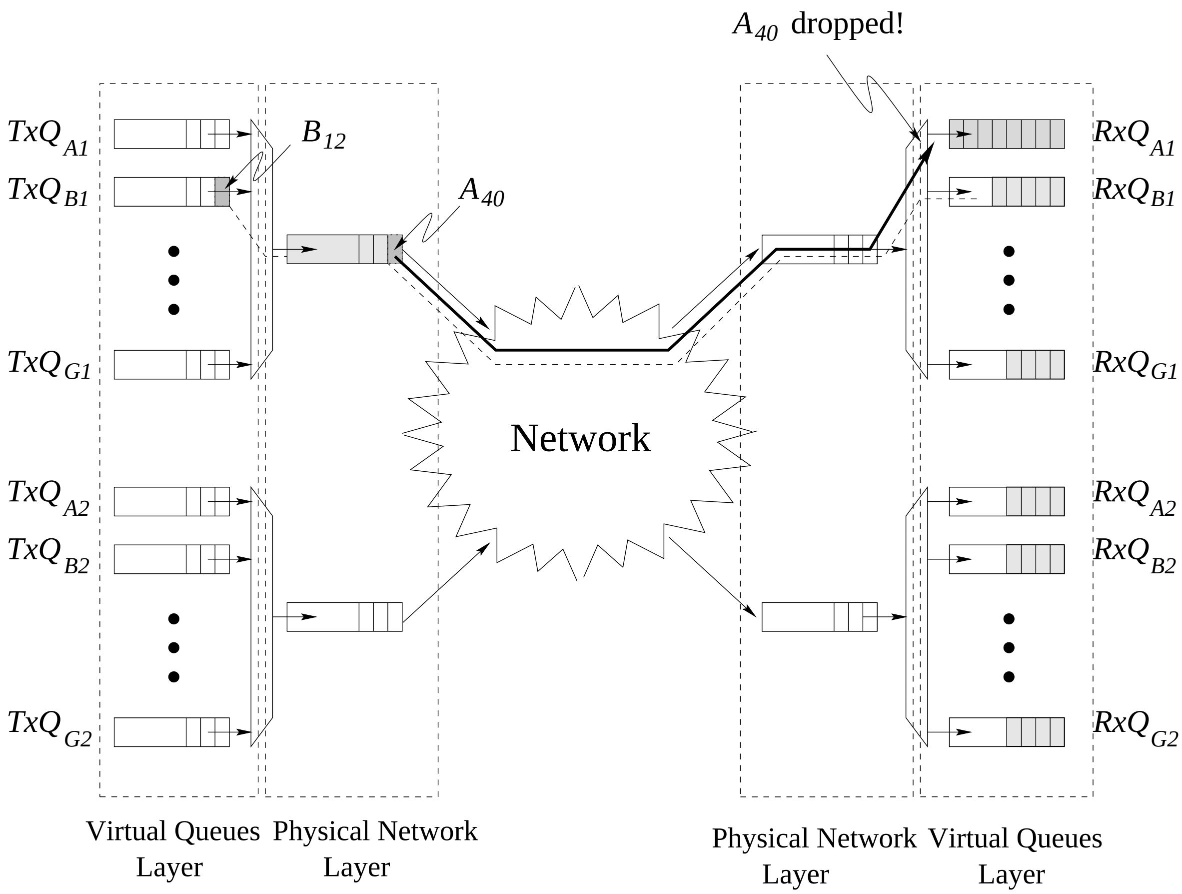


Figure 3-3: A sliding window style loss recovery protocol at the Physical Network layer is insufficient for preventing blockage in the receive queue of one communication domain from blocking traffic in another communication domain which shares the same physical network. Repeated dropping of A_{40} blocks B_{12}.

discipline. In shared network environments, this requires an NIU imposed strategy. This is necessary because the system cannot trust every job to respect the communication discipline and sharing the same pool of network buffers can create inter-job.

Dropping message when receiver buffer is unavailable is simple but requires some recovery strategy if reliable communication is desired. Care has to be taken to ensure that the recovery protocol does not introduce its own dependence arcs between unrelated message queues. In particular, if a variant of sliding window protocol is used, an instance of the protocol is required for each pair of Virtual Queues layer transmit and receive message queues. Relying on a shared instance in the Physical Network layer for this recovery will not work as illustrated in Figure 3-3.

Figure 3-3 shows a case where Packet A_{40} arrives to find its destination queue, RxQ_{A1}, full. The packet is dropped and no positive acknowledgement is provided by the sliding window protocol in the Physical Network layer. This causes A_{40} to be re-sent later. Suppose the program which sent A_{40} has a deadlock bug such that it never frees up queue space to accommodate A_{40}. The sliding window protocol will fail to make progress because the failure to deliver A_{40} prevents the protocol from reporting progress. Packets from queues belonging to other protection domains are thus blocked too, *e.g.* B_{12} waiting in TxQ_{B1}. Basically, any sliding window protocol must operate between TxQ and RxQ *in the same layer*.

When the Physical Network layer provides lossless packet delivery, and losses only occur because a full receive queue causes new packets to be dropped, another possible recovery protocol is to provide acknowledge, either positive or negative, for each packet. Under this scheme, the acknowledgements use a logically different network from the one used for normal packets. A negative acknowledgement (NAck) triggers subsequent re-send [41]. Source buffer space has to be pre-allocated before the message is launched into the network, but copying into this buffer can be deferred by requiring NAck to return the entire message. The appeal of this scheme is its simplicity.

Yet another alternate is to take steps to ensure that there is always sufficient buffer space in the receive queues. The onus of ensuring this can be placed on the user or library code running on the aP. A possible manifestation of this idea is to require the message sender to specify destination address to store the message into. Another is to continue to have the NIU allocate destination buffer, but should the user program make a mistake resulting in a message arriving to find no buffer space, the message will be dropped. To aid debugging, the Virtual Queues layer provides a 1-bit history available to user code indicating whether any message has been discarded. A second option is to provide NIU assistance which dynamically and transparently regulates message traffic to avoid running out of receive queue buffer space. We propose one such scheme, which we call Reactive flow-control.

3.2.3 Reactive Flow-control

Reactive flow-control imposes no flow-control overhead when traffic in the network is well behaved while providing a means to contain the damage wrought by ill behaved programs. It is also well suited to our NIU architecture; as shown in the next chapter, it can be implemented very cheaply on our system.

Under Reactive flow-control, each receive queue has *a low and a high water-mark* as shown in Figure 3-4. When the number of packets in a receive queue is below the high water-mark, the receive queue operates without any flow-control feedback to the transmit queues which send it messages. When occupancy of the receive queue reaches the high water-mark, flow-control throttling packets are sent to all the transmit queues that are potential message sources to throttle message traffic: the senders are told to stop sending packets to this destination until further notified. The receive queue is said to have *overflowed* and gone into *throttled mode*. To ensure that flow-control packets are not blocked by the congestion that it is attempting to control, one logical network of the Physical Network layer is reserved for flow-control packets only. All normal packets go over the other logical network.

After flow-control throttling has been initiated, packets will continue to arrive for the receive queue until throttle is completely in place. The receive queue must continue to accept these packets. Upon receiving a throttling packet, an NIU uses a *selective disable* capability to prevent the relevant transmit queue from sending packets to the receive queue concerned. Packets to other destinations can, however, continue to be sent out from that receive queue.

Each receive queue also has a low water-mark. A receive queue in throttled mode gets out of the mode when its occupancy drops below this water-mark. At that point, the receive queue's NIU sends throttle lifting flow-control packets to the sources to notify them that the receive queue is ready to accept packets again. To prevent a sudden flood of packets to the receive queue, the throttle lifting phase is done in a controlled fashion, by staggering the destination re-enabling at different sources.

Reactive flow-control requires the receive queue to accept all in-coming packets.

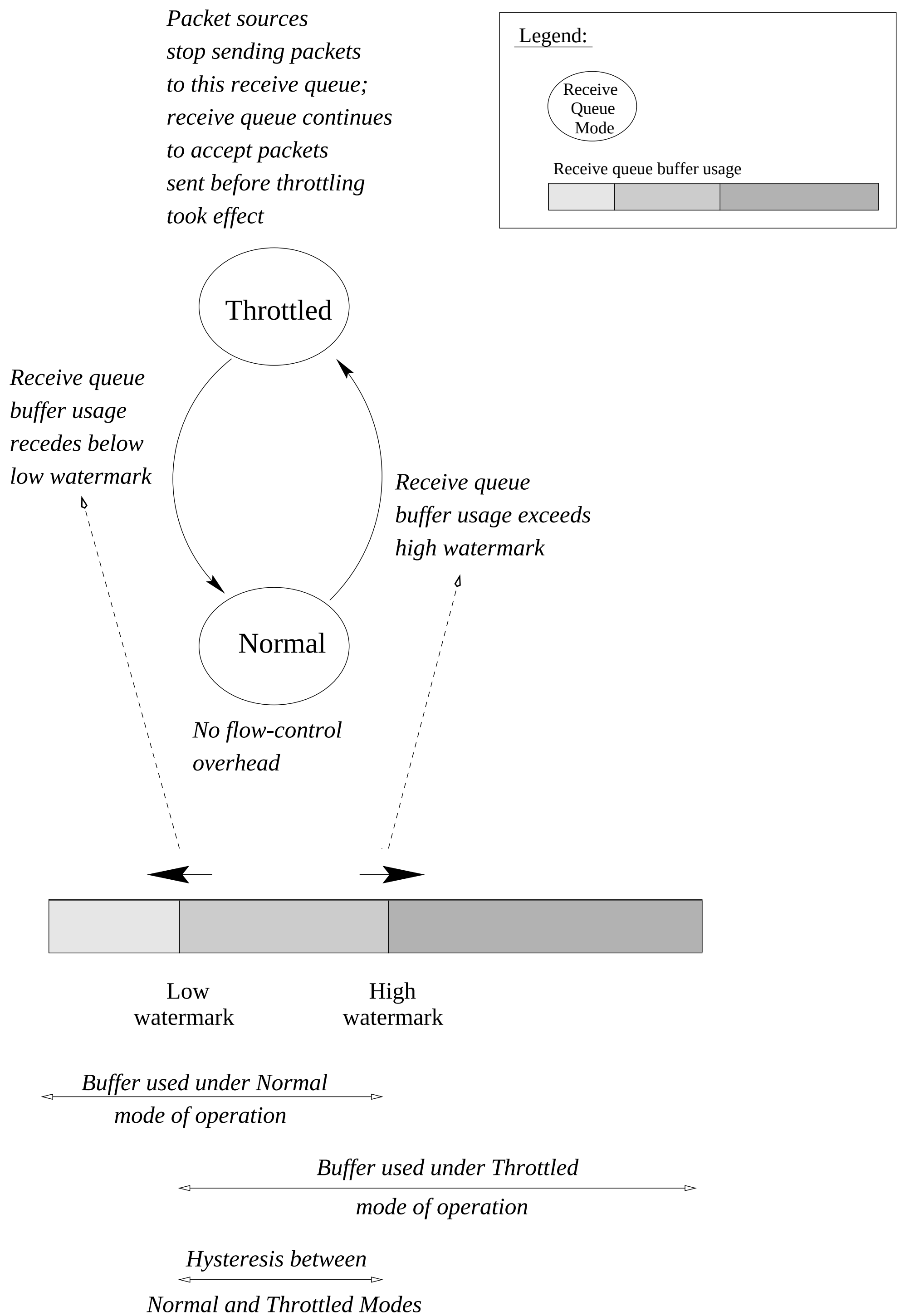


Figure 3-4: Overview of Reactive Flow-control.

For this to work, we must be able to determine the maximum amount of buffering needed. In our architecture, the network layer provides this bound. Both active and passive methods for achieving this bound were discussed in Section 3.1. This amount of buffer space, also known as the *overflow buffer space*, is needed for each receive queue in addition to the *normal buffer space* of size equivalent to the high water-mark. In practice, the overflow buffer size can be substantial (several kilobytes), but because it resides within the owner process's virtual address space, implementations can provide overflow buffering in main memory DRAM, and even locally paged memory.

Strictly speaking, in order to use the network's buffering capacity as a bound on overflow buffer space, as is done in the StarT-Voyager NES described in Chapter 4, a message which crosses the high water-mark should be blocked until the flow-control throttling is known to have taken effect at all the message sources. To avoid this temporary blockage, a *threshold water-mark*, at a point lower than the high water-mark is introduced. Flow-control is initiated when the threshold water-mark is reached, while message blockage pending disabling of all sources is only required when crossing the high water-mark. With an appropriate gap between the threshold and high water-marks, the temporary message blockage can be avoided in most cases.

The water-marks are picked in the following ways:

Low water-mark: The low water-mark should be high enough so that there is sufficient work to keep the receiver busy while throttle is lifted. Thus, if it takes time t_{tl} to remove throttle, the message servicing rate is r_s, and average size of each message is m_s bytes, the low water-mark should be $((t_{tl}/r_s)*m_s)$ bytes. This value should be determined by system code as both t_{tl} and m_s are system parameters. r_s is application dependent, though the system can easily estimate a lower bound; this value can also be specified by the application when it requests allocation of a message queue.

Threshold water-mark: The choice of threshold water-mark is influenced by two considerations. Firstly, the threshold should provide sufficient space for the expected normal buffering requirement of the application. Furthermore, the size between low and threshold water-marks should provide hysteresis to prevent oscillation between throttled and normal mode. For instance, this size should be such that the time to fill it up under maximum message arrival rate is a reasonable multiple (say 8 times) of the time it takes to impose and remove throttle. Hysteresis should work in the other direction too, but we expect message reception service rate to be lower than the arrival rate, so we do not have to worry about throttle being imposed and lifted very quickly. The application should provide system code with the expected buffering requirement, from which the system code determines a threshold water-mark that also has sufficient hysteresis.

High water-mark: The high water-mark is picked so that the size between threshold and high water-marks allows the receiver NIU to continue taking in arriving messages during the time it takes throttle to be put in place. So if it takes time t_{ti} to impose throttle, the message arrival rate is r_a, and average size of each message remains at m_s bytes, the size between threshold and high water-marks should be $((t_{ti}/r_a)*m_s)$. This, like the low water-mark, should be determined by system code. Application code can again assist by estimating the expected r_a.

3.2.4 Decoupled Process Scheduling and Message Queue Activity

To enable more flexible job scheduling policies, message queues remain active independent of the scheduling status of the owner processes. This means messages can be launched into the network from them, and arrive from the network into them while their owner processes are not running. Reactive flow-control takes care of filled receive queues when the receiver process is not active, preventing network deadlock from developing. This will eventually block senders that need to communicate with this receiver. The cluster job scheduler can be informed of receive queue overflow and transmit queue blockage, and use this information to carry out appropriate scheduling actions.

3.2.5 Transparent Process Migration

Using virtual message destination queue names removes one potential obstacle to transparent process migration. When migrating a process which owns a number of receive queues to a new SMP, the global queue names of these receive queues have to be changed. In our design, this simply means changing the virtual to global queue name mapping in the translation tables of the transmit queues that send messages to the affected receive queues. The rest of this section briefly sketches out the communication aspects of job migration.

The system first suspends the migrating process and all processes which communicate with it. Next, changes are made to the transmit queue translation tables, while the network is swept of all outstanding messages to the affected receive queues. Once this is done, the process migration can be completed, and all the suspended processes re-enabled. Transparent migration requires references to other system resources, such as file descriptors, to be migratable too. These issues are, however, othorgonal to communication and beyond the scope of this thesis.

The rest of this discussion describes several features in our design which further reduce the coordination and dead time (*i.e.* time when processes must be suspended) during process migration. Although it is unclear how important these migration cost reductions are compared to other costs of migration, such as copying the migrated process state to the new SMP, the improvements rely on features that are present for other purposes, and thus come "for free".

Improvement 1: Each message transmit and receive queue in the Virtual Queues layer can be disabled individually. When a message transmit queue is disabled, messages will not be launched from it into the network layer. The owner process can, however, continue to enqueue message into the transmit queue, as long as there is buffer space. When a message receive queue is disabled, messages arriving into this queue will not be enqueued into it. Instead, system code handles such messages.

The capability for system code to disable message queues is critical for the Reactive flow-control scheme but also comes in handy for process migration. Instead of suspending processes that may send messages to the migrating process, the system only needs to disable the affected transmit queues until the migration is completed. The processes themselves can continue execution. Queue disabling involves only actions in the NIU's and is a very cheap operation in contrast to process suspension.

Improvement 2: Further improvement to process migration is possible with transmit queue's *selective disable* feature. As the name suggests, applying selective disable to a transmit queue prevents messages heading towards *specific destinations* from being launched from that queue. Messages to other destinations are unaffected. This is, again, motivated by Reactive flow-control considerations, but is useful for relaxing the restrictions on sender processes during process migration.

Improvement 3: Thus far, our description of process migration requires disabling all transmit queues until the network is swept of messages previously sent from them. This is necessary to preserve message ordering. In cases where message ordering is unimportant, these sweeps are unnecessary. To deal with messages heading to a receive queue that is being migrated, a temporary message forwarding service, implemented by the NIU, forwards these messages to the queue's new location. This results in an implementation of process migration without global coordination or global synchronization requirements.

3.2.6 Dynamic Computation Resource Adjustment

The contraction and subsequent re-expansion of the number of processors allocated to a parallel job is likely to be a useful tool for job scheduling. Such changes may be triggered by a cluster's work-load fluctuations or fault induced cluster size contraction.

We will consider two cases here to illustrate how our queue name translation model assists these contractions and re-expansions.

Our first example considers a static parallel execution model with coarse grain dependence, for example, bulk synchronous scientific parallel programs. Because dependence is coarse grain, it is sufficient to implement contraction by migrating some processes so that the parallel job's processes fit into fewer SMP's. This assumes that dependence between processes mapped onto the same processors are coarse grain enough that satisfying the dependence does not require excessive context switching. Re-expansion again simply involves process migration. Using migration capabilities described earlier, both contraction and re-expansion are logically transparent to application code.

Our second example considers a multi-threaded execution model, characterized by dynamically created medium to short threads, and finer grain dependence. Execution, orchestrated by a run time system (RTS), is driven off continuation queues or stacks. Typically, each process has one continuation queue/stack, and work stealing is commonly used for distributing work among processes.

Cid [81] and Cilk [9] are examples of parallel programming systems which fall into this model. Because of fine-grain dependences, an efficient implementation of contraction should combine processes, and the continuation queue/stack of these processes. This is to avoid constant process switches to satisfy fine grain dependences between threads in different processes that are mapped onto the same processor. Message receive queues of these processes should also be combined to improve polling efficiency and timely servicing of messages.

Appropriate RTS support is critical for making such a contraction possible. Our destination queue name translation assists the collapsing of queues by allowing two or more virtual destination queues to be mapped to the same physical queue. With this support, references to (virtual) destination queues can be stored in application data structure before the contraction and continue to be valid after it. Messages sent before the contraction and received after that can also contain references to (virtual) destination queues without encountering problem.

3.3 Application Interface Layer: Message Passing

The third and last layer of this communication architecture is the Application Interface layer, which corresponds to the NIU's interface on the SMP host's memory bus. The service exported by this layer is visible to application code, and is divided between a message passing interface, a shared memory interface, and provision for interface extensions. This section covers the message passing interface.

Software overhead, latency, and bandwidth are several considerations for message transmission and reception [22]. StarT-Voyager provides five mechanisms that offer performance tradeoffs among these considerations. These are: (i) Basic message, (ii) Express message, (iii) DMA, (iv) Express-TagOn, and (v) Basic-TagOn. Custom message passing interfaces can be added using the interface extension capability of the NIU.

For most commercial microprocessors and SMP's, memory mapping is the main interface to the NES. Because neither the memory hierarchy's control structures nor data path are designed for message passing, the interaction between the processor and the NIU has to be constructed out of existing memory load/store oriented operations.

On the transmit side, this interaction has three logical components: (i) determine whether there is sufficient transmit buffer, (ii) indicate the content of the message, and (iii) indicate that the message is ready for transmission. The interaction on the receive side also has three logical components: (i) determine if any message has arrived and is awaiting processing, (ii) obtain message data, and (iii) free up buffer space occupied by the message. Designing efficient methods to achieve these interactions requires taking into account the optimal operating mode of the memory bus, the cache-line transfer mechanism, and the memory model.

With today's SMP's, the number of bus transactions involved in each message send and receive is a primary determinant of message passing performance. Processor book-keeping overhead tends to have a smaller impact on bandwidth and latency because today's processor clock frequency is typically 4 to 7 times the bus clock frequency, whereas each bus transaction occupies the bus for several bus clocks. The consumption of bus bandwidth becomes an even more significant issue in an SMP environment where the memory bus is shared by a number of processors. Several techniques useful for reducing the number of message passing bus transactions are described below.

Use cache-line burst transfers: Virtually all modern microprocessor buses are optimized for cache-line burst transfers. Consider the 60X bus [4] used by the PowerPC 604; a cache-line (32-bytes) of data can be transferred in as few as 6 bus cycles compared to 32 bus cycles required for eight uncached 4-Byte transfers[1]. Aside from superior bus occupancy, the cache-line burst transfer also uses processor store-buffers more efficiently, reducing processor stalls due to store-buffer overflow.

Using burst transfers adds some complexity to NIU design because burst transfers are typically only available for cacheable memory. With message queues read and written by both the processor and the NIU, cache coherence must be maintained between the NES and processor caches in order to exploit the burst transfer. This issue is discussed further when we describe Basic Message support below.

Avoid main memory DRAM: The message data path should avoid going through main memory DRAM in the common case because information that goes through DRAM crosses the system bus twice, first to be written into main memory, and a second time to be read out. This not only wastes bus bandwidth, but increases communication latency; in today's systems, the write-read delay through DRAM can easily add another 15 to 20 bus clocks. Instead, transfers should be directly between the NIU and processor, through any in-line caches between them.

[1] A few microprocessors, such as the MIPS R10000 and Pentium Pro, are able to aggregate uncached memory writes to *contiguous* addresses into larger units for transfer over the memory bus; others offer special block transfer operations, e.g. 64byte load/store instructions in Sparc. These are, however, still non-standard. The PowerPC family used in this project has neither of these features.

Main memory DRAM does have the advantage of offering large amount of relatively cheap memory. An NIU may want to take advantage of it as back-up message buffer space. This is quite easy if the SMP system's bus protocol supports *cache-to-cache transfers*. The feature allows a cache with modified data to supply it directly to another cache without first writing it back to main memory. In such systems, message buffers can logically reside in main memory but be burst transferred directly between the processor cache and the NES.

Bundle control and data into the same bus transaction: Control information can be placed on the same cache-line as message data so that it is transferred in the same burst transaction. For instance, a Full/Empty field can be used to indicate the status of a message. For transmit buffers, the NIU indicates that a buffer is available by setting its Full/Empty field to Empty, and processor software sets it to Full to indicate that the message should be transmitted.

Control information can also be conveyed implicitly by the very presence of a bus event. Because the number of bus events on cached addresses is not directly controlled by software but is dependent on cache behavior, this technique can only be used on uncached addresses. It is employed in our Express and Express-TagOn message passing mechanisms. In those cases, we also make use of address bits to convey "data" to the NIU.

Compress message into single software-controllable bus transaction: This is an advantage if the SMP system uses one of the weak memory models. Found in many modern microprocessor families, weak memory models allow memory operations executed on one processor to appear out-of-order to other devices. In such systems, a message-launch operation might get re-ordered to appear on the bus before the corresponding message-compose memory operations. To enforce ordering, processors that implement a weak memory model provide a *memory barrier* operation (*e.g.* SYNC in PowerPC processors) which is needed between the message-composition operations and the message-launch operation. Unfortunately, memory barrier operations result in bus transactions, and their implementations often wait for superscalar processor pipeline to "serialize".

If the entire content of a message can be compressed into a single software-controllable bus transaction, memory-barrier instructions are avoided. Express message takes advantage of this.

Aggregate Control Operations: An interface that exchanges producer and consumer pointers to a circular buffer to coordinate buffer usage allows aggregation of control operations. Instead of finding out whether there is another free transmit buffer or another message waiting to be serviced, the processor software can find out the *number* of transmit buffers still available, or the *number* of messages still waiting to be serviced. Software can similarly aggregate the "transmit message" or the "free received message buffer space" indicators. Aggregation in these cases, unfortunately, does have the negative side-effect of delaying transmission and the release of unneeded buffer space. There is thus a tradeoff between overhead and latency.

Another form of aggregation is packing several pieces of control information into the data exchanged in one transaction. This can again potentially save bus transactions. Both aggregation techniques are used in our Basic Message support.

Cache Control Information: For control variables that are expected to have good run-length, *i.e.* one party reads it many times before the other writes it, or one party writes it many times before the other reads it, making the location cacheable is advantageous. To actually exchange information, invalidation based snoopy coherence operations will cause at least two bus operations: one to acquire write ownership, and another to read data. Therefore, this is only a good idea if average run-length is greater than two.

A possible application of this idea is to cache consumer pointer of receive queue as is done in the CNI message passing interface [77]. This pointer is written frequently by processor software but typically read infrequently by the NIU.

Similarly, the consumer pointer of transmit queue is also a candidate, although in that case making it uncached has its merits. This is because to obtain long write run-length on the NIU side for a transmit queue consumer pointer, the processor software has to use a local variable to keep track of the number of available buffers, and only read the consumer pointer from the NIU to update this variable when it reaches zero. That being the case, making the consumer pointer uncached is actually better because there is no need for the NIU to obtain write ownership as is necessary if the location is cached.

In the case of receive queue consumer pointer, making the pointer uncached and using aggregation to reduce the number of write bus transactions has the negative side-effect of delaying buffer release. A cached consumer pointer allows the write to occur to the processor cache, which only goes on the bus when the NIU needs it.

Special Support for Multicast/Broadcast: An NIU interface that allows multicast or broadcast operations to be specified can reduce the number of times the same data is moved over the memory bus. Our TagOn message capitalizes on this, while giving the flexibility for some of the message data to be different for each "multicast" destination.

One of the biggest challenges in improving message passing performance is to reduce the latency of reading information into the processor. Unfortunately, there are few ways of improving this other than to reduce the number of such occurrences using techniques described above. Invalidation based snoopy bus protocol provides no opportunity to push data into caches unless the processor has requested for it. Snoopy bus protocol that allow update or snarfing [6], neither of which are found in commercial systems today, may open up new opportunities. Processor software can also use pre-fetching to hide this latency, but this is not always easy or possible, particularly since the data can become stale if the pre-fetch occurs too early.

We studied the above design options in 1994 when we designed the StarT-NG [19], and again in late 1995 when we started designing StarT-Voyager. Independently, Mukherjee et al [77, 78] also studied the problem of message passing interface design for NIU connecting to coherent memory buses. The most interesting result of their work is a class of message passing interfaces that they named Coherent Network Interfaces (CNI). We will discuss these after we describe Basic Message. This section describes the message passing mechanisms in our Application Interface layer.

3.3.1 Basic Message

The Basic message mechanism provides direct access to the Virtual Queues layer's messaging service. With a 32-bit header specifying the logical destination queue and other options, and a variable payload of between four and twenty-two 4-byte words, a Basic Message is ideal for communicating an RPC request, or any medium size transfer of up to several hundred kilobytes.

The Basic Message interface consists of separate transmit and receive queues, each with a cacheable message buffer region, and *uncached* producer and consumer pointers for exchanging control information between the processor and the NIU. Status information from the NIU — transmit queues' consumer pointers and receive queues' producer pointers — are packed into a 4 byte value so that they can all be obtained with a single read. The message buffer region is arranged as a circular FIFO with the whole queue visible to application software[2], enabling concurrent access to multiple messages. The message content is specified by value, *i.e.* the processor is responsible for assembling the message content into the transmit buffer space. An uncached pointer update immediately triggers NIU processing.

The processor performs four steps to send a basic message (Figure 3-5, top half). The Basic Message transmit code first checks to see if there is sufficient buffer space to send the message (Step 1). That figure also shows several messages that were composed earlier, and are waiting to be transmitted. When there is buffer space, the message is stored into the next available buffer location (Step 2); the buffer

[2]This is not strictly true in that the NIU maintains an overflow buffer extension for each receive queue where incoming messages are temporarily buffered when the normal, software visible receive queue is full.

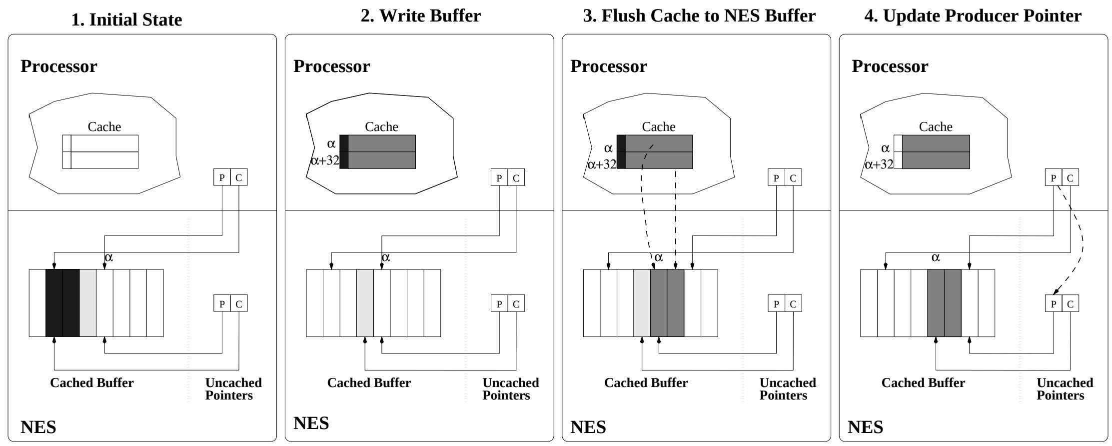


Figure 3-5: Sending a Basic Message

is maintained as a circular queue of a fixed but configurable size. The transmit and receive queue buffers are mapped into cached regions of memory. Unless *NIU Reclaim* mode is used, in which case the NIU is responsible for buffer space coherence maintenance, the processor must issue CLEAN instructions to write the modified cache-lines to the corresponding NIU buffer locations (step 3). For systems with weak memory model, a barrier instruction is required after the CLEAN instructions and before the producer pointer is updated via an uncached write. This write (step 4) prompts the NIU to launch the message, after which the NIU frees the buffer space by incrementing the consumer pointer.

The application processor overhead can be reduced by using the NIU Reclaim facility where the NIU issues CLEAN bus operations to maintain coherence between the processor cache and the NIU buffers. In this case, the pointer update will cause the NIU to reclaim the message and then launch it.

Though the transmit and receive queues are mapped to cached regions, producer and consumer pointers are mapped to *uncached* regions to ensure that the most up-to-date copies are seen both by the application and the NIU. To minimize the frequency of reading these pointers from the NIU, software maintains a copy of the producer and consumer pointers (P, C in top half of figure). The copy of the consumer pointer needs to be updated only when it indicates that the queue is full; space may have freed up since by then. The NIU may move the consumer pointer any time it launches a messages, as illustrated in our example between steps 1 and 2.

Message reception by polling is expected to be the common case, although an application can request for an interrupt upon message arrival. This choice is available to a receiver on a per receive queue basis, or to a sender on a per message basis. When polling for messages, an application compares the producer and consumer pointers to determine the presence of messages. Messages are read directly from the message buffer region. Coherence maintenance is again needed so that the application does not read a cached copy of an old message. As before, this can be done either explicitly by the processor with FLUSH instructions or by NIU Reclaim.

A unique aspect of the Basic Message buffer queue is its memory allocation scheme. Buffer space in this queue is allocated in cache-line granularity and both the producer and consumer pointers are cache-line address pointers. Allocation in smaller granularity is undesirable because of the coherence problem caused by multiple messages sharing a cache-line. The other obvious choice of allocating maximum-sized buffers was rejected because it does not work well with either software pre-fetching of received messages, or NIU Reclaim. The main problem is that the size of a message is unknown until the header is read. Therefore, both pre-fetching and a simple implementation of NIU Reclaim must either first read the header, and then decide how many more cache-lines of data to read, or blindly read all three cache-lines. The former introduces latency while the latter wastes bandwidth. With cache-line granularity allocation, every cache-line contains useful data, and data that is fetched will either be used for the current message, or subsequent ones. Better buffer space utilization is another benefit of this choice.

When we designed the Basic message support, we considered both the producer-consumer pointer scheme described above, and a scheme which uses Full/Empty bits in each fixed-size buffer. (We adopted the latter scheme in StarT-NG [19], the predecessor of StarT-Voyager.) Tables 3.1 and 3.2 compare the bus transaction costs of the two schemes.

A Full/Empty bit scheme is expected to have better performance because the control functions do not incur additional bus transactions. Under the producer-consumer pointer scheme, the cost of obtaining free transmit buffers can probably be amortized and is thus insignificant. Similarly, the cost of releasing receive buffers should be insignificant. But this still leaves this scheme at a disadvantage because of the extra cost of indicating transmit and finding out about received messages, both of which can only be amortized at a cost to end-to-end latency.

We choose the producer-consumer pointer based scheme despite its performance disadvantage due to implementation considerations. Section 3.3.6 provides a more detailed comparison of implementation complexity of pointer and full-empty bit handshake schemes.

3.3.2 Express Message

Messages with a minimal amount of data are common in many applications for synchronization or communicating a simple request or reply. Basic Messages, with a cached message buffer space, is a bad match because the bandwidth of burst transfer is not needed while the overhead of coherence maintenance, weak memory model (if applicable), and explicit handshake remains. Express Messages are introduced to cater to such small payloads by utilizing a single uncached access to transfer all the data of a message and thus avoid these overheads of Basic Messages.

A major challenge of the Express Message design is to maximize the amount of data transported by a message while keeping each compose and launch to a single, uncached memory access. The Express Message Mechanism packs the transmit queue ID, message destination and 5 bits of data into the *address* of an uncached write. The NIU automatically transforms the information contained in the address into a message header and appends the data from the uncached write to form a message. Figure 3-6 shows a simplified format for sending and receiving Express Messages. Additional address bits can be used to convey more information but they consume larger (virtual and physical) address space and can also have a detrimental effect on TLB if the information encoded into the address bits does not exhibit “good locality”. (Alternate translation mechanisms such as PowerPC’s block-address translation

Item	Description	Cost and Frequency
Transmit		
T1	Read transmit queue consumer pointer	1 bus transaction (aggregatable)
T2	Write message content	1 to 2 bus transactions each cache-line. Only 1 bus transaction to move data from processor cache to NIU for each cache-line if it is already present in cache with write permission. Otherwise cache-miss or ownership acquisition incurs another bus transaction.
T3	1 memory barrier operation	1 bus transaction (aggregatable)
T4	Write transmit queue producer pointer	1 bus transaction (aggregatable)
	Total (n cache-lines message; not using Reclaim) b: average number of buffers obtained at each read of transmit queue consumer pointer. s: average number of buffers sent at each write of transmit queue producer pointer.	$(n + 1/b + 2/s)$ to $(2n + 1/b + 2/s)$
Receive		
R1	Read receive queue producer pointer	1 bus transaction (aggregatable)
R2	Read message content	1 bus transaction each cache-line.
R3	Removal of stale data	1 bus transaction each cache-line.
R4	1 memory barrier operation	1 bus transaction (aggregatable)
R5	Write receive queue consumer pointer	1 bus transaction (aggregatable)
	Total (n cache-lines message) r: average number of buffers obtained at each read of receive queue producer pointer. f: average number of buffers freed at each write of receive queue consumer pointer.	$(2n + 1/r + 2/f)$

Table 3.1: This table summarizes the bus transaction cost of Basic Message support, assuming NIU Reclaim is not used. With NIU Reclaim, the receive case incurs an additional bus transaction for each cache-line of data. NIU Reclaim for the transmit case does not incur any additional bus transaction if the snoopy bus protocol supports cache-to-cache transfer. Otherwise, it also incurs an additional bus transaction for each cache-line of data.

Item	Description	Cost and Frequency
Transmit		
T1	Read next transmit buffer's Full/Empty bit.	1 bus transaction
T2	Write message content	1 additional bus transaction to move data from processor cache to NIU for the cache-line which contains the Full/Empty bit (assuming that the read in T1 also obtained write permission); 1 to 2 bus transactions for each additional cache-line. Only 1 bus transaction to move data from processor cache to NIU for each cache-line if it is already present in cache with write permission. Otherwise cache-miss or ownership acquisition incurs another bus transaction.
T3	1 memory barrier operation	1 bus transaction
T4	Write Full/Empty bit	0 additional bus transactions.
	Total (n cache-lines message; not using Reclaim)	$(n + 2)$ to $(2n + 1)$
Receive		
R1	Read next receive buffer's Full/Empty bit.	1 bus transaction
R2	Read message content	0 additional bus transactions for the cache-line which contains the Full/Empty bit; 1 bus transaction for each additional cache-line.
R3	Removal of stale data	1 bus transaction for each cache-line after the first one.
R4	1 memory barrier operation	1 bus transaction
R5	Write Full/Empty bit	1 additional bus transaction, assuming the earlier read of the cache-line has obtained write permission. This bus transaction accounts for the NIU reading this bit.
	Total (n cache-lines message; not using Reclaim)	$(2n + 1)$

Table 3.2: This table summarizes the bus transaction cost of a Full/Empty bit based scheme, assuming software is responsible for coherence maintenance. If something analogous to NIU Reclaim is used, each transmitted or received cache-line incurs an additional bus transaction.

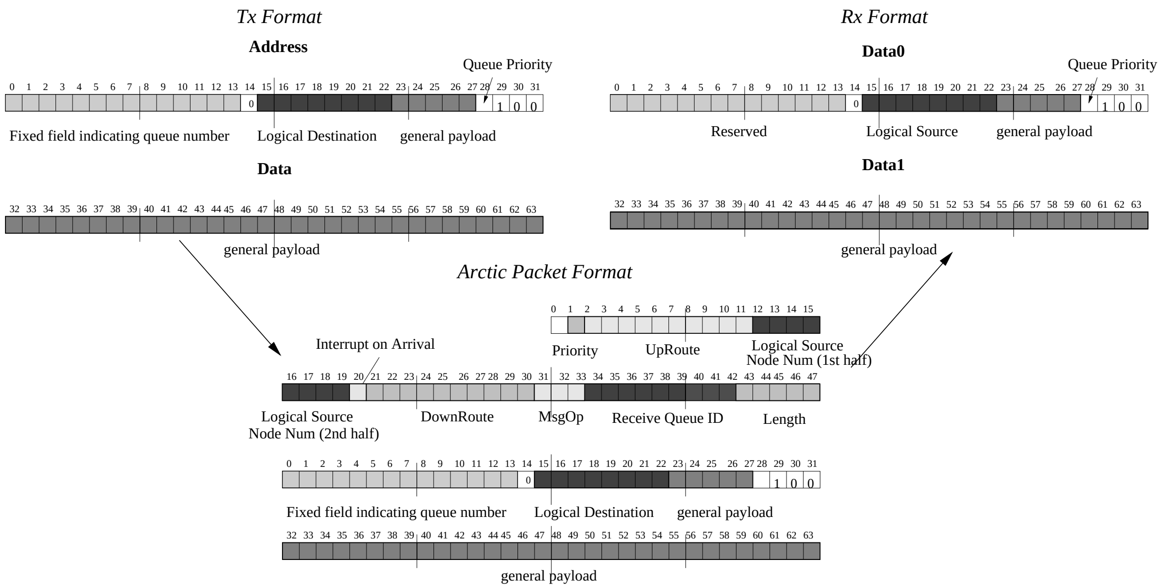


Figure 3-6: Express Message Formats

mechanism[72] may be employed to mitigate this problem but this depends on both processor architecture and OS support.) Producer and consumer pointers are still required for transmit queues as a mechanism for software to find out the amount of buffer space available in the transmit queues.

To reduce the data read by a receive handler, the NIU reformats a received Express Message packet into a 64 bit value as illustrated in Figure 3-6. Unlike Express Message transmits, address bits cannot be used to convey message data to the processor when receiving an Express Message. With processors which support double-word read into contiguous GPR's, an Express Message receive is accomplished with a 64 bit uncached read. For processor families without this support but with 64 bit floating points, an Express Message receive can be accomplished with a 64 bit uncached read into an FPR, and the data is subsequently move into GPRs. Alternatively, two 32 bit loads into GPRs can be issued to receive the message.

Unlike Basic Messages, the addresses for accessing Express Message queues do not specify particular queue entries. Instead, the NIU provides a FIFO push/pop interface to transmit and receive Express Messages. Due to this side effect, speculative loads from Express Message receive regions are disabled by setting the page attributes

Item	Description	Cost and Frequency
Transmit		
T1	indicate transmit and content	1 bus transaction
Receive		
R1	poll received message	2 bus transactions if using 32-bit uncached read; 1 bus transaction if using 64-bit uncached read;

Table 3.3: Bus transaction cost of Express message transmit and receive.

appropriately. When an application attempts to receive a message from an empty receive queue, a special Empty Express Message, whose content is programmable by system code, is returned. If message handler information is encoded in the message, such as in Active Messages[100], the Empty Message can be treated as a legitimate message with a "no action" message handler.

The Express message interface is implemented as a veneer on top of circular buffer queues controlled with producer-consumer pointers. The NIU automatically increments these pointers in response to read/write bus transactions. Software can actually directly access the lower-level interface in the same way as for Basic message, except that up to 4 messages now packs into each cache-line. When there are a number of Express messages to send (*e.g.* multicasting to say 8 destinations), it is cheaper to use this lower-level interface (3-6 bus transactions for the 8 messages, compared to 8 bus transactions. In systems with larger cache-line size, this advantage is even greater.)

Although using the lower-level interface can be useful on the receive end too, software has to guess whether there is going to be many messages for it to receive, and then select the interface. This can be difficult. The NIU, which knows the number of messages waiting to be processed in each Express RxQ, can help by making this decision for software. (This feature is not in the current version of the StarT-Voyager described in Chapter 4.) When an RxQ has a large number of messages pending, the NIU response to a poll not with the next message, but with a special message that includes the identity of the Express RxQ, and the current pointer values; this is currently already used for One-poll on Basic messages. Software can then switch to using the lower-level interface to receive Express messages. The advantage of using the lower-level interface for receiver is even greater than for transmit, since on StarT-Voyager we use two 32-bit uncached read (plus a possible SYNC) to receive each Express message. Using the lower-level interface, two cache-line of 8 messages can be received with 4-5 bus transactions, compared to 16 with the usual interface.

3.3.3 DMA Transfer

DMA support provides an efficient mechanism for moving contiguously located data from the memory on one node to that on another. It resembles traditional DMA facility in that an aP can unilaterally achieve the movement without the remote aP's involvement. This is in contrast to data movement effected through normal message passing, where matching request and reply messages are needed. The DMA facility can be thought of as a remote memory get or memory put operation. DMA requests are specified with virtual addresses, allowing zero-copying message passing to be implemented. When global shared memory address space is involved, the DMA guarantees *local coherence* [58] but not global coherence. This means that the data that is read or written is coherent with any cache copies in the source and destination nodes respectively, but not necessarily with cache copies elsewhere in the system. Kubiatowicz [58] provides arguments for this model of DMA in shared memory machines.

The DMA facility is designed to be "light weight" so that it can be profitably employed for relatively small sized transfers. To reduce per-transfer aP overhead, the design decouples page pinning from transfer request. The cost of the former can be amortized over several transfers if traditional system calls are used to pin pages. Alternatively, system software can be designed so that the aP Virtual Memory Manager cooperates with the sP, allowing the latter to effect any necessary page pinning. User code initiates DMA transfers through an interface similar to a Basic Message transmit queue. The transfer direction, *logical* source and destination, source data (virtual) address, destination data (virtual) address, and length are specified in a DMA request message to the local sP. This sP, together with the other sP involved in the transfer, performs the necessary translation and protection checks before setting up DMA hardware to carry out the transfer. An option for messages to be delivered after DMA completion, *e.g.* to notify the receiver of the sender, is also available.

3.3.4 TagOn Message

The Express-TagOn Message mechanism extends the Express Message mechanism to allow additional data located in special NIU memory to be appended to an out-going message. The Express-TagOn Message mechanism was designed to eliminate a copy if message data was already in NIU memory. It is especially useful for implementing coherent shared memory protocol, and for multi-casting a medium sized message. As composed by an application, an Express-TagOn Message looks similar to an Express Message with the addition that several previously unused address bits now specify the NIU memory location where the additional 32 Bytes (or 64 Bytes) of message data can be found. For protection and to reduce the number of bits used to specify this location, the address used is the offset from a TagOn base address. This base address can be different for each transmit queue, and is programmed by system code.

At the destination NIU, an Express-TagOn Message is partitioned into two parts that are placed into two separate queues. The first part is its header which is delivered like an Express Message, via a queue that appears to be a hardware FIFO. The second part, made up of the data that is "tagged on", is placed in a separate buffer similar to a Basic Message receive queue which utilizes explicit buffer deallocation.

Express-TagOn Messages have the advantage of decoupling the header from the message data, allowing them to be located in non-contiguous addresses. This is useful in coherence protocol when shipping a cache-line of data from one site to another. Suppose the sP is responding to another site's request for data. This is achieved by the sP first issuing a command to move the data from aP-DRAM into NIU memory, followed by a Express-TagOn Message that ships the data to the requester. In addition to the cache-line of data, an Express-TagOn Message inherits the 37 bit payload of

Express Messages which can be used in this instance to identify the message type and the address of the cache-line that is shipped. Cache-line data may also be brought into the NIU without the sP asking for it, for example the aP's cache may initiated a write-back of dirty data. In such cases, Express-TagOn Message's ability to decouple message header and data allows the data to be shipped out without further copying.

Express-TagOn Messages are also useful for multi-casting. To multi-cast some data, an application first moves it into NIU memory. Once that is done, the application can send it to multiple destinations very cheaply, using a Express-TagOn Message for each one. Thus, data is moved over the system memory bus only once at the source site, and the incremental cost for each destination is an uncached write to indicate a Express-TagOn Message.

Basic-TagOn extends Basic Messages in a way similar to Express-TagOn. It differs in that the non-TagOn part of the message can contain a variable amount of data. Furthermore, when a Basic-TagOn message is received, it is not separated into two parts, but is instead placed into one receive queue, just like an ordinary Basic Message. Basic TagOn message offers similar advantages as Express-TagOn at the transmit end: separation of message body permitting more efficient multi-cast.

Basic-TagOn was added fairly late in the design. It was mostly added to make the support uniform: TagOn is an orthogonal option available with both Basic and Express messages.

3.3.5 One-poll

In order to minimize the overhead of polling from multiple receive queues, StarT-Voyager introduces a novel mechanism, called *OnePoll*, which allows one polling action to poll simultaneously from a number of Express Message receive queues as well as Basic Message receive queues. A single uncached read specifies within some of its address bits the queues from which to poll. The result of the read is the highest priority Express Message. If the highest priority non-empty queue is a Basic Message queue, a special Express Message that includes the Basic Message queue name and its queue pointers is returned. If there are no messages in any of the polled queues, a special Empty Express Message is returned.

OnePoll is useful to user applications, most of which are expected to have four receive queues: Basic and Express/Express-TagOn, each with two priorities. The sP has nine queues to poll; clearly the OnePoll mechanism dramatically reduces the sP's polling costs.

3.3.6 Implications of Handshake Alternatives

The Full/Empty bit scheme requires the control portion of the NIU to poll and write locations in the message buffers. With message queue buffer usage occurring in FIFO order, the NIU knows the buffer in each queue to poll on next. This section considers several implementations; invariably, each is more complex than those for the Basic message mechanism.

First consider a simple implementation, where the NIU provides message buffer SRAM and the control portion of the NIU reads the next buffer's Full/Empty bit field from this SRAM. The NIU control also writes to the SRAM to set or clear Full/Empty bits. Both actions consume SRAM memory port bandwidth, a contention problem that increases in severity if the same SRAM port is used for other functions such as buffering received messages. The polling overhead also increases as the number of queues supported in the NIU increases. Even without considering contention, the time to perform this polling increases with the number of message queues as the number of message buffer SRAM ports is unlikely to increase at the same rate.

Clearly, to implement the Full/Empty scheme efficiently, the NIU has to be smarter and poll only when necessary. This requires snooping on writes to message buffer space, and only polling after writes have occurred. If NIU Reclaim is supported, the snooping will only reveal an attempt to write, due to acquisition of cache-line ownership. The NIU should then Reclaim the cache-line after a suitable delay. The design also has to deal with the situation where the NIU is unable to transmit as fast as messages are composed. In order that it does not drop any needed poll, the NIU has to keeps track of the message queues with pending polls, probably in a condensed form. A design that blocks write or write-ownership acquisition until the NIU can transmit enough messages to free up hardware resources to track new message transmit requests is unacceptable as it can cause deadlocks.

The contention effect of NIU reading Full/Empty bits can be circumvented by duplicating the slice of the message buffer SRAM where the Full/Empty information is kept, devoting it to NIU Full/Empty bits polling. Using the same idea to reduce the contention effect of NIU writing Full/Empty bits requires slightly more complex data paths. Because the system bus width is unlikely to be the same width as the message buffer size, certain bits of the data bus takes data either from the Full/Empty SRAM or the normal message data SRAM, depending on which part of the message being driven on to the bus. A mux is thus needed. Making this scheme work for variable sized message buffers will further increase complexity. The most likely design in that case is to not only store the value of Full/Empty bit, but also store a bit to determine if that location is currently used for that purpose.

If we constrain software to modify the Full/Empty bit in a FIFO order, and assume fixed-size message buffers, the above design can be further simplified. The NIU need not poll on the Full/Empty bits. Instead, it simply snoops on the *data* portion of a bus transaction in addition to the address and control parts. When it sees a set Full/Empty field in a bus transaction to a region associated with a transmit queue, it goes ahead to increment that transmit queue's producer pointer. Similarly, a cleared Full/Empty bit for a bus transaction associated with a receive queue triggers increment of that queue's consumer pointer. The constraint that the Full/Empty bit of message queues has to be set or cleared in FIFO order is unlikely to be a problem if each message queue is used by only one thread. If several threads share a message queue, it may incur additional coordination overhead.

Although the Full/Empty bit design is well understood, the implementation constraints we faced in StarT-Voyager make it infeasible. Because we are using off-the-shelf dual-ported SRAM's for message buffers, with one port of the SRAM directly connecting to the SMP system bus, we are unable to use the duplicated memory slice idea to remove the overhead of NIU writing Full/Empty bits. Even duplicating the slice to remove NIU Full/Empty read contention is not seriously considered because of concerns about capacitance loading on the SMP system bus[3]. We would have picked the Full/Empty bit design had we been implementing the NIU in an ASIC with much more flexibility over the data-path organization.

The implementation of the producer-consumer pointer scheme is much simpler. One reason is that the data and control information are clearly separated. In fact, software updates message queue pointers by providing the new values of the pointer in the *address* portion of a memory operation. We also allow both read and write operations to update the pointers, with the read operation returning packed pointer information from the NIU.

The separation of data and control makes it feasible to implement producer and consumer pointers in registers located with control logic. By comparing these pointers, the control logic determines whether the queue is full or empty and whether the NIU needs to carry out any action.

3.3.7 Comparison with Coherent Network Interface

The Coherence Network Interfaces (CNI's) [77] use a combination of consumer pointers and Full/Empty bits to exchange control information between processor software and the NIU. The pointers are placed in cached address locations to take advantage of expected long write run-lengths.

To avoid explicitly setting the Full/Empty bits to empty, they added a clever idea of sense-reverse on the Full/Empty bits – the meaning of a 1 in the Full/Empty bit field changes as the usage of the queue reaches the end of the linear buffer address region and wraps around to the beginning. Thus, whereas in one pass, 1 indicates Full buffers, the value indicates Empty buffers on the next pass. Using this scheme requires fixed-size message buffers and the linear buffer region to be a multiple of that fixed size. The hardware complexity of implementing CNI is around that of the generic Full/Empty bit scheme. While it dispenses with the NIU clearing transmit queue Full/Empty bits, it adds the need to maintain full cache-coherence on the

[3] This was a serious concern earlier in the design when we were targeting a 50MHz clock rates. With our final clock rate of 35MHz, this should not be an issue.

producer pointers.

The bus transaction costs of using CNI is shown in Table 3.4. It actually looks very similar to the generic Full/Empty bit scheme. A definitive comparison of CNI with the Full/Empty bit scheme can only be done with a specific system, taking into account a number of details, including processor's memory system details and actual NIU implementation. For instance, under CNI, software does not read the Full/Empty bit to determine if a transmit buffer is available, so that round-trip read latency is avoided. However, software using a generic Full/Empty bit scheme can hide this cost by pre-fetching the next buffer's Full/Empty bit when it starts to use the current transmit buffer. CNI's use of a cached receive queue consumer pointer is a good idea. Under the reasonable assumption that the receive queue does not run out of space very frequently it is better than using an uncached consumer pointer, or using Full/Empty bit to indicate release of receive queue buffer.

3.4 Application Interface Layer: Shared Memory

The shared memory interface implements both CC-NUMA and S-COMA support. The NIU's behavior on the system bus differs in the two cases. It is the slave for bus transactions generated by CC-NUMA cache misses, and has to carry out actions at remote sites to obtain data or ownership. In response to requests from other nodes, the NIU must be able to behave as a proxy bus master, fetching data from local main memory, or forcing it from local caches. The NIU also has to keep directory information for those cache-lines for which it is the "home site".

In clusters with multiprocessor SMP nodes, CC-NUMA implementation presents some anomaly if the host SMP does not support cache-to-cache data transfer of both clean and dirty data. Most snoopy bus protocols do not require, nor allow a cache to respond to a read transaction on the bus if it has the requested data in clean state. Thus, it is possible that a remotely fetched cache-line, d_a is present in processor A of an SMP, but processor B encounters a cache miss of the same cache line which has to be serviced by the SMP's NIU. For simplicity of the distributed CCDSM protocol,

Item	Description	Cost and Frequency
Transmit		
T1	Read transmit queue consumer pointer	1 bus transaction (aggregatable)
T2	Write message content	1 to 2 bus transactions each cache-line. Only 1 bus transaction to move data from processor cache to NIU for each cache-line if it is already present in cache with write permission. Otherwise cache-miss or ownership acquisition incurs another bus transaction.
T3	1 memory barrier operation	1 bus transaction
T4	Write Full/Empty bit	0 additional bus transactions.
	Total (n cache-lines message; not using Reclaim) b: average number of buffers obtained at each read of transmit queue consumer pointer.	$(n + 1 + 1/b)$ to $(2n + 1 + 1/b)$
Receive		
R1	Read next receive buffer's Full/Empty bit.	1 bus transaction
R2	Read message content	1 bus transaction each cache-line.
R3	Removal of stale data	1 bus transaction each cache-line.
R4	1 memory barrier operation	1 bus transaction
R5	Write receive queue consumer pointer	0 bus transaction if hit in cache; 2 bus transactions each time NIU actually reads consumer pointer.
	Total (n cache-lines message; not using Reclaim) h: average number of times software writes the consumer pointer before the NIU reads it.	$(2n + 1 + 2/h)$

Table 3.4: This table summarizes the bus transaction cost of a CNI style scheme, assuming software is responsible for coherence maintenance for receive queues. If something analogous to NIU Reclaim is used on the receive queues, each received cache-line incurs an additional bus transaction.

the NIU should implement a cache of remotely fetched data so that processor B's cache-miss can be serviced with data from this cache. This may be implemented with NIU firmware.

NIU support for S-COMA keeps coherence state information for cache-lines in local main memory, and snoops on bus transactions addressing these regions. Cases where further actions at remote nodes are needed use infrastructure similar to those for implementing CC-NUMA support. A further requirement is the ability to translate address between local main memory address and global shared memory address. NIU S-COMA support should include hardware snooping, so that the sP is not involved when the current cache-line state permits the bus transaction to complete. Otherwise, the advantage of S-COMA over CC-NUMA is badly eroded.

The shared memory interface includes provision for passing hints from application code to the NIU. This can be implemented fairly simply by using a memory mapped interface to addresses interpreted by the sP.

The problem of designing and implementing correct cache-coherence protocol is still a major research area today [73, 21, 89, 90, 26, 86, 87]. Deadlock aside, implementing a coherence protocol that actually meets the specifications of its memory model is also a difficult problem. Part of the problem is that many memory models are motivated by implementation convenience and specified in an operational manner that is often imprecise and incomplete. It is beyond the scope of this thesis to delve into the problem of coherence protocol design, implementation and verification. Instead, we target an NIU design which leaves details of the coherence protocol programmable. This both decouples the final details of coherence protocols from NIU hardware design and implementation, and permits future experimentation.

Providing the ability to program the coherence protocol takes care of logical errors. In addition, hard-wired low-level resource sharing policies in our NIU design must not can cause deadlock. This problem is actually protocol dependent; whether a certain partitioning of resources is adequate safe-guard against resource sharing induced deadlock for a protocol is dependent on the *dependence chains* that can arise under the protocol. One solution is to restrict the types of *dependence chains* that a protocol can create, and constrain protocols to be written in a way that respect these limitations. This approach is adopted for PP code in FLASH's MAGIC chip [60], and is applicable to our design. A second possibility, which our design also supports, is to provide the capability to extend the number of resource pools through firmware.

3.5 Support for Interface Extensions

The NIU firmware programmable core has several capabilities that enable extending the application communication interface. These capabilities, described below, are in addition to a basic instruction set encompassing general integer and control instructions, and access to a reasonable amount of memory.

The NIU firmware observes the system bus to detect both explicit and implicit initiation of communication. This is achieved through bus slave and bus snooper capabilities. The NIU firmware is the slave for a static region of physical address space. Write bus transactions to this address space are forwarded to NIU firmware, while read bus transactions are supplied data by the NIU firmware.

The NIU also snoops on a static region of main memory. NIU firmware can specify, at cache-line granularity, the type of bus transactions it is interested in *observing*, and those that it is interested in *intervening*. In the latter case, the bus transaction is not allowed to complete until NIU firmware gives the approval, and optionally provide data to read-like transactions.

NIU firmware is able to initiate any bus transaction to an arbitrary physical address. If data is transferred, the NIU specifies the memory location on the NIU to transfer the data to or from, including regions in the transmit and receive queues. This, together with NIU firmware's ability to directly read and write these NIU memory locations enables NIU firmware to indirectly read and modify the SMP's main memory locations.

General message passing capability is available to NIU firmware. A TagOn-like capability makes it easy for NIU firmware to send out data that is in NIU SRAM. The message passing capability also allows physical addressing of message destination. In this way, NIU firmware can implement destination translation.

NIU firmware has sufficient capability to generate network packets of arbitrary format. Thus, the firmware message passing capability not only enables NIU firmware on different nodes to communicate, but allows NIU firmware to generate network packets that are processed completely by NIU hardware at the destination. The NIU firmware can also intercept selected in-coming packets and take over their processing. This selection is based on a field in the packet header, normally inserted during destination translation at the message source.

All the NIU firmware interface extension capabilities are composable. They function as an instruction set that NIU firmware uses to deliver the functions of new communication interfaces. Because NIU firmware is limited to one or a small number of threads, but is multiplexed between many different functions, it is important that its execution is never blocked at the hardware level. Instead, all blockage must be visible to the firmware so that it can suspend processing the affected request, and switch to processing other requests. The latter may be necessary to clear up the blockage.

Chapter 4

StarT-Voyager NES Micro-architecture

This chapter describes the StarT-Voyager Network Endpoint Subsystem (NES), an NIU that connects IBM RISCSystem/6000 Model 43P-240 SMP's to the Arctic network [12, 13]. The IBM RISCSystem/6000 Model 43P-240 SMP, also called Doral, was first introduced at the end of the fourth quarter of 1996. With two processor card slots and two PCI I/O buses, it is a desktop class machine marketed as an engineering graphics workstation. Each processor card contains a PowerPC 604e running at 167MHz and a 512 kByte in-line L2 cache. The system bus conforms to the 60X bus protocol [4] and runs at 66MHz in the original system. In StarT-Voyager, we replace one of the processor cards with our NES. Figure 4-1 shows both the original Doral SMP and the one used in the StarT-Voyager system.

We would have preferred to use larger SMP's, but real life constraints concerning access to technical information, system cost and timely availability of the SMP led to our choice. An earlier iteration of this project, the StarT-NG [19], targeted an eight-processor, PowerPC 620 based SMP that was being developed by Bull. Unfortunately, the PowerPC 620 microprocessor was never fully debugged or sold commercially. Naturally, the SMP we were targeting did not materialize either.

The demise of that project taught us a few lessons. In StarT-NG, strong emphasis was placed on the absolute performance of the resulting system. To meet this goal,

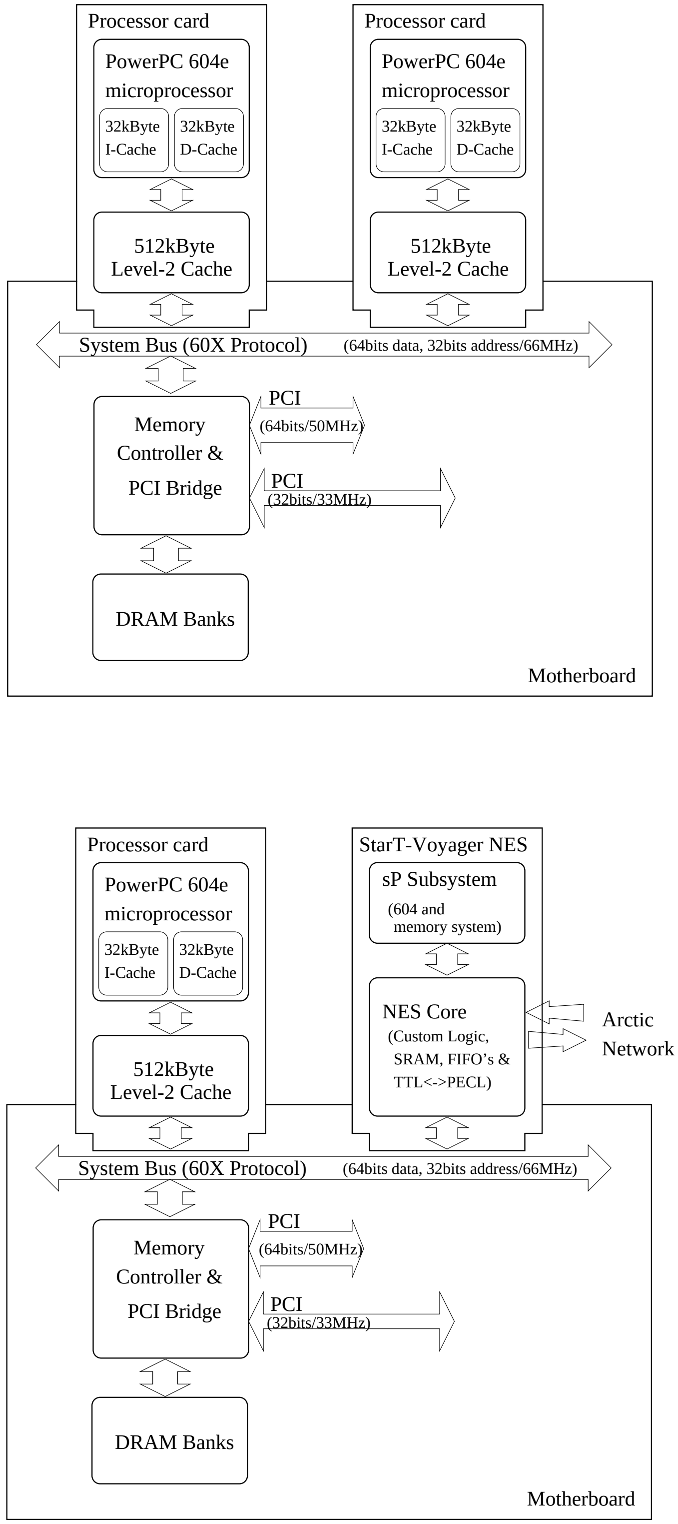


Figure 4-1: The top of the diagram shows an original IBM Doral SMP. The bottom shows a Doral used in the StarT-Voyager system, with a processor card replaced by the StarT-Voyager NES.

we targeted projected commercial systems that were still in very early stages of development. This was to ensure that the completed cluster system would be available around the same time as the host SMP, rather than much later as a "late" system using previous generation SMP's. This choice greatly increased the risk of the chosen host SMP system not materializing, and is an unnecessary risk for architecture research. Although the absolute clock speed of StarT-Voyager is lower than contemporary systems, both its architecture at the microprocessor and SMP system levels, and its relative clock ratios are highly similar to today's systems. Our research results are therefore directly applicable. For example, the newest PowerPC microprocessor from IBM, the 750, announced in September 1998, has a micro-architecture almost identical to that of the 604e; the only differences being the presence of a back-side L2 cache interface on the 750 and a much higher processor core clock rate of 450 MHz.

Once we were convinced that absolute clock speed was no longer a high priority for our project, we decided to implement the custom portions of the NES with FPGA's (Field Programmable Gate Arrays) from Xilinx and an LPGA (Laser Programmable Gate Array) from ChipExpress. This reduces both the manpower and financial costs, while improving the architecture research potential of the system as FPGA allows relatively easy hardware design modifications after the NES is constructed. It also reduces the risk of the project. Although we had access to IBM technical manuals of the devices on the system bus, we were unable to get simulation models from IBM for verifying our design. Instead, our hardware design is verified in a simulation system that uses device models we wrote based on our reading of the manuals. Using FPGA's reduces the risk from mis-interpreting the manuals as changes can be made during bring-up if such problems are detected. The price for using FPGA and LPGA technologies is a lower NES and memory bus clock frequency of 35MHz.

Host SMP Memory System Characteristics

The PowerPC 604e implements a weak memory model, where memory accesses are not guaranteed to appear in program order to external devices. When such ordering is important, it is necessary to insert SYNC instructions. Each SYNC instruction

State	Read okay?	write okay?	dirty?
Modified (M)	Yes	Yes	Yes
Exclusive (E)	Yes	Yes	No
Shared (S)	Yes	No	No
Invalid (I)	No	No	No

Table 4.1: Semantics of the four cache-line states in MESI coherence protocol.

ensures that memory access instructions before it are completed and visible to external devices before those after it.

The 60X bus protocol uses the MESI cache coherence protocol, an invalidation based protocol with caches maintaining a 2-bit state for each (32 byte) cache-line, representing one of the four states: (i) M: Modified, (ii) E: Exclusive, (iii) S: Shared, and (iv) I: Invalid. Table 4.1 describes the semantics of these states. The PowerPC processor family provides the Load-reserve/Conditional-store (LR/SC) pair of instructions as the atomicity primitives.

The 60X bus protocol does not support cache-to-cache transfers, *i.e.*, data is not normally supplied from one cache to another, even if one cache has a dirty copy of a cache-line that another cache wants to read. Instead, the dirty cache-line is first written back to main memory before it is read out again. The Doral system provided an exception to this: it can accommodate an L2 look-aside cache which can intervene and act like main memory, supplying data to read transactions and accepting data from write transactions. This limited form of cache-to-cache transfer capability is utilized by the NES.

Arctic Network Characteristics

The Arctic network is a high performance, packet switched, Fat-Tree network. It supports variable packet sizes between 16 to 96 bytes in increments of 8 bytes. This packet size includes a 6 byte packet header and a 2 byte CRC. Packet delivery is reliable, *i.e.* lossless, and in-order for packets using the same up-route through the Fat-tree. Except for nearest neighbor nodes, multiple paths through the network exist between each source-destination node pair. Each link of the Arctic network delivers a

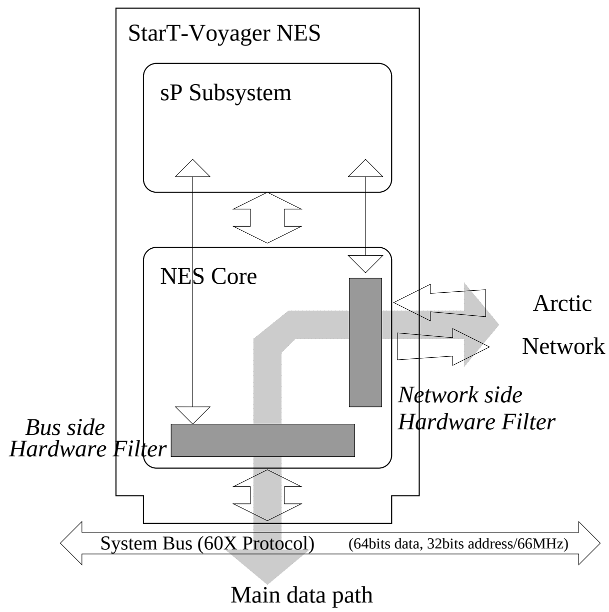


Figure 4-2: A high level view of the StarT-Voyager Network Endpoint Subsystem (NES). The hardware filters determine whether the sP should be involved in processing a particular event. In most cases, NES Core hardware handles the events completely, ensuring high performance. The sP handles corner cases and provides extendibility.

bandwidth of 150 MBytes per second. Two links, an in link and an out link, connects to each SMP in the StarT-Voyager system.

Arctic distinguishes between two priorities of packets, high and low, with high priority ones having precedence over low priority packets when routing through a switch. Furthermore, switch buffer allocation is done in such a way that the last buffer is always reserved for high priority packets. Effectively, this means that high priority packets can always get through the network, even if low priority ones are blocked. The converse is, however, not true. Therefore, strictly speaking, Arctic does not support two fully independent networks, but the design is adequate for request-reply style network usage, where one network (reply/high) must remain unclogged even when the other (request/low) is blocked, but not vice-versa.

4.1 StarT-Voyager NES Overview

The NES design faces the challenge of delivering a wide range of functions at high performance while providing programmability. Furthermore, it must be implementable with moderate effort. We meet these demands with the design shown in Figure 4-2, which couples a moderate amount of custom hardware, the NES Core, with an off-the-shelf microprocessor, a PowerPC 604 microprocessor which we refer to as the sP (service processor). The idea is to have custom hardware completely handle the most frequent operations, so that the sP is not involved very frequently. Because the sP takes care of infrequent corner cases, the custom hardware can be kept simple. Conversely because the custom hardware completely takes care of most cases, overall performance is little affected even if the sP is somewhat slower at handling operations. This makes it feasible to employ a normal off-the-shelf microprocessor to provide NES programmability.

Using an Off-the-Shelf Service Processor

Compared to a custom designed programmable engine, a generic microprocessor reduces the design effort, but faces the problems of *inadequate capabilities* and *slow access to off-chip devices* due to deep pipelining between its high-speed processor core and the external bus. To overcome these deficiencies, we designed the *custom logic in the NES Core as a co-processor to the sP*, offering it a flexible set of composable, communication oriented commands. These commands provide missing functions and off-loads data movement tasks that are simple for the NES core to provide but are inefficient for the sP to undertake.

The strength of the sP is its ability to implement complex control decisions that can also be modified easily. Thus a guiding principle in the NES design is to have the sP make control decisions which are carried out by the NES Core. With this organization, most of the communication data move between the network and the host SMP system through the NES Core, by-passing the sP. This design philosophy is also adopted in the FLASH's MAGIC chip [60], where the PP typically does not handle data directly, but only deals with control information such as network packet headers.

A disadvantage of firmware implemented functions is lower throughput, since handling each event takes multiple firmware instructions spanning a number of processor cycles. With today's commercial microprocessors, which are all single-threaded, this limits concurrency. A custom designed programmable core suffers similar inefficiencies although it could opt to support multiple threads simultaneously to improve throughput. In contrast, dedicated hardware for handling the same event can be pipelined so that several operations can simultaneously be in progress. Custom dedicated hardware can also exploit greater parallelism, *e.g.* CAM (Content Addressable Memory) hardware for parallel lookup.

As before, this problem is partly resolved with the communication primitives supplied by the NES Core hardware. With an appropriate set of primitives, and an efficient interface between the sP and the NES Core, the sP can simply specify the sequence of actions needed, but it neither carries out those actions by itself, nor closely supervises the ordering of the sequence. This reduces the occupancy of the sP. A detailed discussion of the interface between sP and NES Core can be found in Section 4.4.

The sP has its own memory subsystem consisting of a memory controller and normal page-mode DRAM. The decision to provide sP with its own memory system, as opposed to using the host SMP's main memory, is discussed in the next section (Section 4.2), where several alternative organizations are presented.

Overview of NES Hardware Functions

The NES Core provides full hardware implementation of Basic message, Express message and their TagOn variants. These message passing services are available to both the aP *and the sP*. Because the buffer space and control state of these message queues both reside in the NES Core, only a fixed, small number of hardware message queues are available – 16 transmit and 16 receive queues. To meet our goal of supporting a large number of queues, additional message queues, with their buffer and control state residing in DRAM, are implemented with the assistance of the sP.

Both hardware implemented and sP implemented message queues present identical interfaces to aP software. Switching a logical message queue used by aP software between hardware and sP implemented queues involves copying its state, and changing VMM mapping on the aP. This can be done in a manner completely transparent to aP software. The switch is also a local decision. The NES Core maintains a TLB-like hardware which identifies the RQID's associated with the hardware receive queues. A packet with an RQID that misses in this lookup is handled by the sP.

NES hardware handles the repetitive operations in a DMA. At the sender, NES Core hardware reads data from system bus, packetizes and sends them into the network. At the receiver, NES Core hardware writes the data into DRAM, and keeps count of the number of packets that have arrived. The sP at both the source and the destination are involved in non-repetitive operations of a DMA transfer such as address translation, and issuing commands to the functional units that perform the repetitive tasks. (See Section 4.5.7 for details.)

NES Core hardware also implements cache-line state bits for main memory falling within S-COMA address space, using them to impose access permission checks on bus transactions to this address space. (We sometimes refer to this as the Snooped Space.) In the good cases where the cache-line is present in a state that permits the attempted bus transaction, the bus transaction completes with no further delay than a cache-miss to local main memory in the original SMP. The sP can ask to be *notified* of such an event. In other cases, the bus transaction is retried until the sP is notified and replies with an approval. When giving approval, the sP can optionally supply the data to a read-like bus transaction. Section 4.5.9 elaborates on S-COMA support.

The sP is directly responsible for another address space, the sP Serviced Space. Typically, write-like bus transactions are allowed to complete, with their control and state information enqueued for subsequent processing by the sP. Read-like bus transactions are typically retried until the sP comes back with approval. The NES Core hardware that supports this actually provides other options, as described later in Section 4.5.8. CC-NUMA style shared memory implementation is one use of this address space. Other uses are possible; because the sP interprets bus transactions to this address space, aP software and sP can impose *arbitrary semantics* to bus transactions to specific addresses.

The sP can also request for bus transactions on the aP bus. These are specified as commands to the NES Core, with the sP specifying all 60X bus control signal values. NES Core execution of these bus transactions is decoupled from the sP's instruction stream processing. (Further details in Section 4.5.6)

Our sP is similar to embedded processors on message passing NIU's like Myrinet and SP2 in that it has the ability to construct, send and receive arbitrary message packets across the network. (See Sections 4.5.1 through 4.5.5.) In addition, the NES Core provides the sP with very general capability not present in those systems to *observe and intervene in aP system bus transactions.* (Details in Sections 4.5.8 and 4.5.9.)

The sP operates as a software functional unit with many virtual functional units time multiplexed on it. NES Core hardware improves the sP's efficiency by implementing *configurable hardware filters*, which only present it with bus transactions or packets that it wants to see or handle. On the bus side, the hardware cache-line state-bit check performs that function. On the network side, the destination translation and receive packet RQID lookup perform this filtering for out-going and in-coming packets respectively.

4.2 Alternate Organizations

This section examines several alternate NES organizations, covering major design choices that fundamentally shape the NES micro-architecture.

4.2.1 Using an SMP processor as NIU Service Processor

An alternative to our StarT-Voyager NES organization is a design which uses one of the SMP processors as the sP (See Figure 4-3). (Typhoon-0 [92] has a similar organization, except that the NES is split into two devices: a device on the system bus to impose fine-grain access control, and a device on the I/O bus for message passing access to the network.) This design has the advantage of not needing an sP

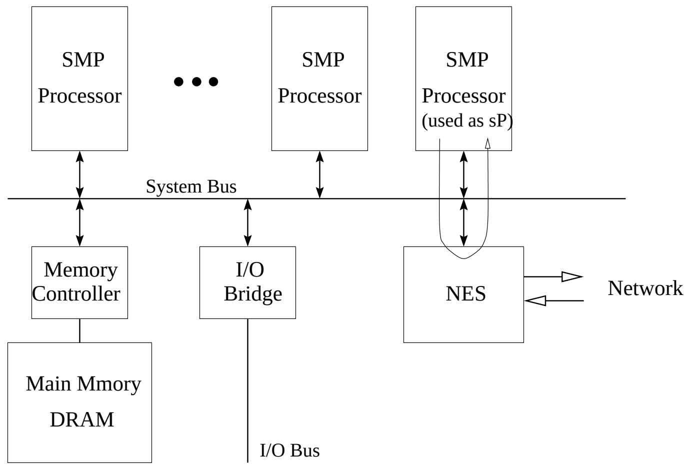


Figure 4-3: A design which uses one of the SMP processors to serve as the NES sP.

subsystem on the NES. Unfortunately, this design has a serious performance problem arising from the sP having to constantly poll the NES to find out if it has any events to handle. This translates into system bus usage, even when there is nothing for the sP to do[1].

There is also the possibility of deadlocks. Part of the sP's function is to be intimately involved in controlling the progress of bus transactions on the system bus. In this capacity, there will be cases when the sP wants to prevent bus transactions from completing until it has carried out some action such as communicating with other nodes. With the sP functions undertaken by one of the SMP processors, these actions now depend on the ability of this sP to use the system bus, either to communicate with the NES or simply to access main memory. Insufficient bus interface resources may prevent such system bus transactions from completing, as elaborated below.

In an SMP, bus interface resources in microprocessor and other system bus devices are divided into separate pools devoted to different operations. This is to avoid deadlocks arising from dependence chains in snoopy cache coherence operations. The specifics of a design, *e.g.* the number of pools of resources and the mapping of bus

[1]Caching device registers, proposed by Mukherjee et al. [77], can remove this bus bandwidth consumption when there is no event to handle. However, it will lengthen the latency of handling events since an actual transfer of information now requires two bus transactions: an invalidation and an actual read.

interface operations to the resource pool, is tied to the dependence chains that can arise under its particular bus protocol.

In bus-based SMP, it is common to assume that a push-out – write transaction triggered by a bus snoop hitting a dirty local cache copy – is the end of a dependence chain. Such a bus transaction will always complete because its destination, memory, is effectively an infinite sink of data.

Read transactions, on the other hand, may depend on a push-out. Consequently resources used for read transactions should be in a different pool from push-out, and dependence from this pool to that used for push-out will build up dynamically from time to time. The fact that push-out transactions do not create new dependence prevents dependence cycles from forming.

This assumption, that a push-out transaction never creates new dependence, is violated when a normal SMP processor, used as an sP, delays the completion of a push-out until it is able to read from main memory. Whether this will really cause a deadlock, and whether there are any acceptable work-arounds, depend on details of the SMP system; *e.g.* whether push-out queues operate strictly in FIFO order, or permits by-passes. Nevertheless, the danger clearly exists.

StarT-NG

In the StarT-NG project, we explored an organization that is very similar to using an SMP processor as the sP (See Figure 4-4). It differs in that the PowerPC 620 processor was designed with a back-side L2 cache interface that can also accommodate a slave device. Our design made use of this capability to connect the NES through a private (*i.e.* non-shared) interface to a 620 processor that also directly connects to the system bus. This processor, used as the sP, would poll the NES via its back-side L2 cache interface rather than the system bus[2]. One way to think about this

[2]The StarT-NG design reported in [19] divides the equivalent of our NES core into two portions. One portion, which is message passing oriented, interfaces to the back-side L2 cache interface. The other portion, which takes care of shared memory oriented functions, is mostly accessible only via the system bus with the exception that notification of pending events is delivered to the sP via the message passing interface. Another difference is that StarT-NG's shared memory support is a subset of StarT-Voyager's.

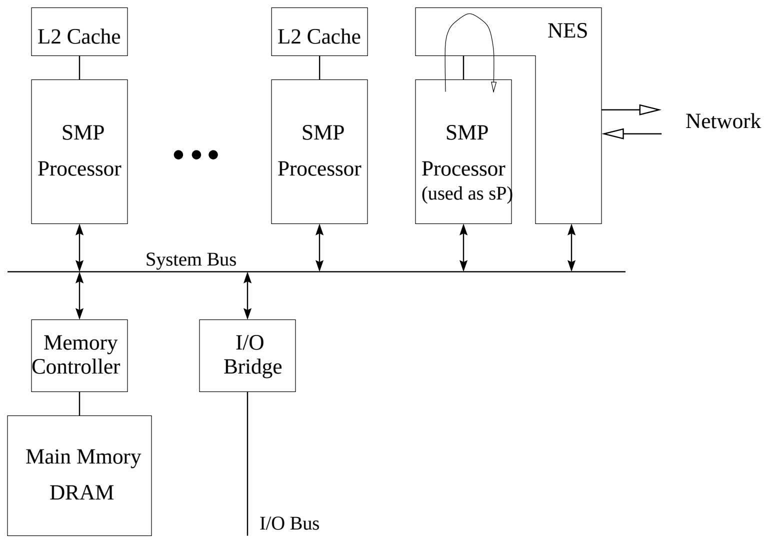


Figure 4-4: The StarT-NG design, which uses one of the SMP processors as the NES sP, but capitalizes on this processor's special back-side L2 cache to have a private interface between the NES and this processor.

is StarT-Voyager's sP bus is moved to the back-side L2 cache interface. This design solves the performance problem of sP constantly polling over the system bus.

To deal with the problem of potential deadlocks from the sP directly using the system bus, usage rules are imposed on sP software in the StarT-NG design. For instance, the sP processor and the other SMP processors cannot share cache-able memory. This avoids any possibility of dependence arising from snoopy coherence operations on the shared data. In addition, many bus transactions that the sP wants performed on the system bus have to be done through the NES Core even though sP software is capable of issuing instructions that result in the desired bus transactions on the system bus. This again is to avoid deadlocks; an example follows.

Consider the case where the sP wishes to flush a particular cache-line from other caches in the SMP node. A Flush bus transaction on the system bus will achieve this. But even though the sP can directly execute a Dcbf (Data Cache Block Flush) instruction that will result in such a bus transaction, it is not safe to do so. The Flush bus transaction may require a push-out from another SMP processor before it completes. In turn, that push-out may be queued behind several other push-outs requiring sP processing. But until the Flush bus transaction triggered by the sP's Dcbf instruction is completed, the sP's instruction pipelined is blocked. A deadlock results[3].

While there are several ways of circumventing this particular deadlock scenario, delegating the task of performing the Flush bus transaction to the NES Core is the only general one[4]. It ensures that the sP's execution is decoupled from the completion of the Flush bus transaction. While waiting for the latter's completion, the sP is free to handle push-outs or any of many other events. Because many functions are multiplexed onto the sP, any scenario which can block its execution pipeline must be closely examined.

Using the SMP main memory as sP memory is a more elegant design than the separate sP memory system adopted in the StarT-Voyager NES. But as shown above, several serious issues have to be resolved before it is feasible. In StarT-Voyager, the lack of a back-side L2 cache interface on the PowerPC 604e processor precludes the solution used in StarT-NG. The next best solution that both avoids dedicated sP DRAM and maintains a private bus between the sP and the NES Core is to have the NES Core act as a bridge chip, relaying sP bus transactions onto the SMP system bus to access main memory. We decided that for our experimental prototype, this was too much trouble; we were able to put together the sP's memory system using commercial parts with little design effort.

[3]This example assumes that items in the push-out queues are serviced *in-order*. The problem does not arise if the queuing policy allows by-passing. Unfortunately, this level of detail is rarely specified in microprocessor manuals. Furthermore, neither a bus based SMP, nor a more traditional hardware implemented cache-coherent distributed shared memory machine requires by-passing in the push-out queue. Since allowing by-pass requires a more complex design, it is unlikely that the feature is implemented.

[4]Other solutions typically rely on implementation details that allow absolute bounds on buffering requirements to be computed and provided for. We view these solutions as too dependent on implementation specifics which should not be tied down rigidly. For example, should the buffer space in the processor bus interface increase because a new version of the chip has more silicon available, this solution will need the amount of resources to be increased.

4.2.2 Custom NIU ASIC with Integrated Programmable Core

Incorporating a custom programmable core into the NIU is an alternative that can not only achieve the goals that we laid out in earlier chapters, but also overcome some disadvantages of an off-the-shelf sP. Aside from having the programmable core closer to the rest of the NIU, this approach brings many opportunities for customizing the programmable core's instruction set, and perhaps adding multiple contexts, or even simultaneous multi-threading support.

A number of projects have taken this approach, *e.g.* FLASH [60], SUN's S3.mp [83, 82] and Sequent's NUMA-Q [68]. Though all three machines contain some kind of custom programmable core, they vary in generality. Most are highly specialized micro-code engines. The MAGIC chip in FLASH is the closest to a generally programmable core.

In the interest of keeping design effort down, we elected not to include a custom designed programmable core. We also felt that with the correct mix of simple hardware and an off-the-shelf processor, good communication performance can be achieved. Furthermore, the higher core clock speed of our sP may allow it to perform more complex decisions on the occasions when it is involved. In contrast, PP, the programmable core of MAGIC, is involved in all transactions. The number of cycles it can spend on handling each event is therefore constrained in order to maintain adequate throughput and latency. This can become a particularly serious issue with multiple processor SMP nodes[5].

4.2.3 Table-driven Protocol Engines

Using hardware protocol engines driven by configurable tables offers some amount of programmability over how a cache coherence protocol operates without employing micro-code engine or general programmable core. The flexibility is of course limited to what is configurable in the tables. Since it is less general, its implementation is

[5]Each MAGIC chip serves only one single R10000 processor in FLASH.

likely to retain the low latency and high throughput advantage of hardwired coherence protocol engines. Its implementation is also simpler than a generally programmable core.

The StarT-Voyager NES actually employs this idea in a limited way. As explained later in Section 4.5.9, the NES Core performs cache-line granularity access permission checks for system bus transactions addressed to a portion of main memory DRAM. (This memory region can be used to implement S-COMA, and local portion of CC-NUMA shared memory.) The outcome of this hardware check is determined not only by cache-line state information, but also by configurable tables which specify how the cache-line state information is interpreted. Useful as a component of our overall design, the table driven FSM approach alone does not provide the level of flexibility that we want in our NIU.

4.3 StarT-Voyager NES Execution Model

The NES Core is unusual as a co-processor in that its execution is driven by multiple external sources of stimuli triggering concurrent execution in shared functional units. These sources are: (i) commands from the sP, (ii) packets arrival from the network, and (iii) bus transactions on the SMP system bus. Internally, the NES Core consists of a collection of functional units connected by queues. Execution proceeds in a continuation passing/dataflow style, with each functional unit taking requests from one or more input queues and in some cases, generating (continuation) results into output queues. These queues, in turn, feed other functional units.

This model of functional units connected by queues extends across the entire cluster, with requests from one NES traveling across the network into the queues of another NES. A very important part of the design is to ensure that dependence cycles do not build up across this vast web of queues.

Although the NES Core has several sources of stimuli, the notion of per-thread context is weak, limited to address base/bound and producer/consumer pointer values that define FIFO queues. In the case of transmit queues, the context also includes the destination translation table and base/bound addresses which define the NES SRAM memory space used for TagOn messages.

Conspicuously absent is a register file associated with each command stream. Unlike register style instructions used in RISC microprocessors today, but similar to tokens in dataflow machines, the commands in the NES Core specify operand information by value. This choice is motivated by hardware simplicity. Each of the sP command streams can probably make use of a small register file, but since most events handled by the sP are expected to result in only a small number of commands, the context provided by a register file in the NES Core is not critical[6]. Without the register file and the associated register operands fetching, NES Core hardware is simplified.

Management of ordering and dependence between commands from the same stream, discussed in Section 4.4, is also kept very simple. There is no general hardware support for dynamic data dependence tracking, such as score-boarding. A few special commands are, however, designed to allow vector-processor style chaining (See implementation of DMA in Section 4.5.7).

4.4 Interface between sP and NES Custom Functional Units

The interface between the sP and the NES Core is critical to performance. sP occupancy and the overall latency of functions implemented by the sP can vary significantly depending on this interface. In designing this interface, we have to keep in mind that it is relatively expensive for the sP to read from the NES Core. It also takes a fair number of cycles for the sP to write to the NES Core. Three possible interface designs are discussed here: (i) a traditional command and status register

[6]We will see later in the evaluation chapter, Chapter 5 that the cost of switching context is a significant overhead for sP firmware. That suggests that multiple context support in the sP is likely to be useful. However, that is not the same as saying that the NES Core should support multiple contexts for its command streams. Firmware must be able to get to those contexts cheaply if it is to be an advantage.

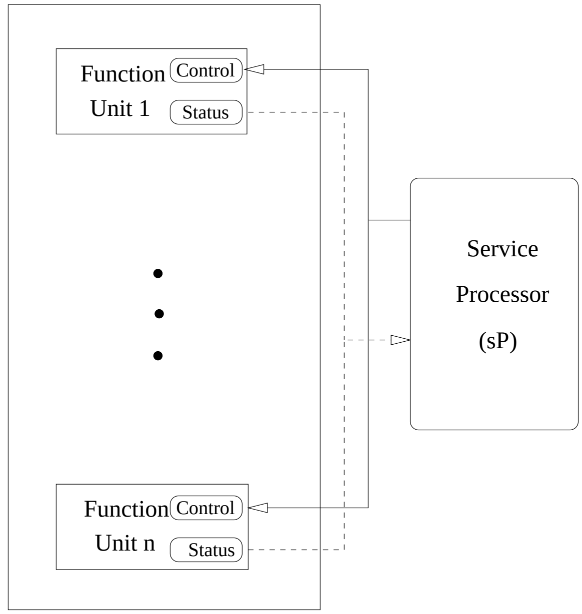


Figure 4-5: A command and status register interface.

interface common among I/O devices; (ii) a command and completion queues design with ordering guarantee between selected commands; and (iii) an interface similar to (ii) with the addition of *command templates*. The StarT-Voyager NES implemented the second option.

4.4.1 Option 1: Status and Command Registers

I/O devices are commonly designed with status and command registers which are memory mapped by the microprocessor. As illustrated in Figure 4-5, the microprocessor issues requests by writing to command registers and checks the results or progress of these requests by reading from status registers. The I/O device can usually also notify the microprocessor of an event with an interrupt but this is expensive.

Traditional I/O device interfaces are not designed for efficiency as microprocessors are expected to access them infrequently. A major limitation of this interface is each command register typically supports only one request at a time. The microprocessor has to withhold subsequent requests to the same command register until the current command is completed.

Secondly, when a sequence of operations to different registers (*i.e.* different functional units) of the device contains dependences, the required ordering has to be enforced externally by the microprocessor. The microprocessor has to delay issuing a command until commands which it depends on are known to have completed.

Thirdly, polling individual status registers to find out about progress or completion of a command is slow. This can be improved by packing status bits together so that a single poll returns more information. There is, however, a limit to the amount of information that can be packed into a single access; because status reads are normally done with uncached bus transactions, this limit is typically 32 or 64 bits. Checking through a vector of packed status bits can also be slow for the sP firmware.

An sP working with such an interface will be very inefficient. To accomplish a sequence of commands, the sP has to enforce inter-command dependence and arbitrate between functional unit usage conflicts. In the mean time, it has to poll for both completion of previous requests, and arrival of new transactions to handle. Performance is degraded because polling is relatively slow and pipelining between NES Core command execution and sP issue of subsequent requests is severely limits by the poll-and-issue sequence. The sP code also becomes very complex because it has to choreograph one or more sequences of actions while servicing new requests arriving from the network or the SMP bus[7].

4.4.2 Option 2: Command and Completion Queues

Ideally, the sP would like to issue, all at once, an entire sequence of commands needed for processing a new request. This simplifies sP coding by removing the need to continually monitor the progress of the sequence. It also facilitates pipelining: not only can the sP issue more commands while earlier ones are being executed, the NES Core can potentially exploit parallelism between these commands.

The efficiency of sP polling for new requests or for notification of command completion can be improved by merging them into polling from a single address. The

[7]Interleaving processing of multiple transactions is not merely a performance improvement option but a necessity as failure to continue servicing new requests can lead to deadlocks.

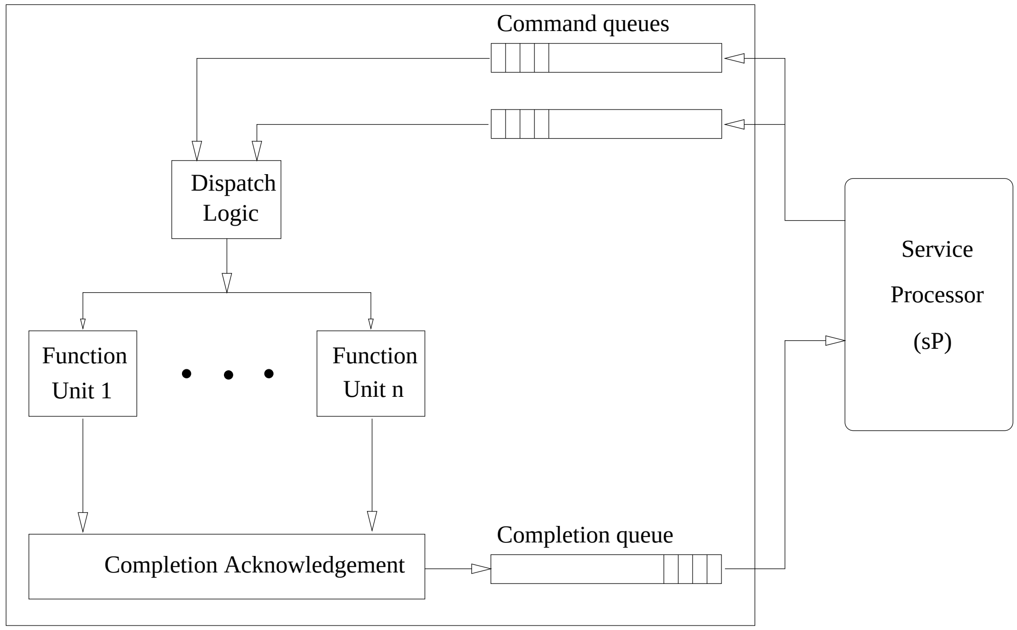


Figure 4-6: A command and completion queues interface.

StarT-Voyager NES achieves these advantages with an interface composed of two command queues, a completion queue and the ability to perform OnePoll from several queues.

4.4.2.1 Command Ordering and Data Dependence

A major command queue design issue is hardware maintenance of ordering and data dependence between commands. Without such guarantees, the sP cannot fully exploit the command queues since it still has to manually enforce data dependence in the old way. In addressing this issue, a design has to balance among the need for ordering, exploitation of inter-command parallelism and design complexity. The following are some options we considered.

Dependence Tracking Hardware: At one extreme is an aggressive design that dynamically tracks dependence between commands due to read/write access to shared NES SRAM locations. Several implementations of this dependence tracking are possible. One is to employ score-boarding, with full/empty bits for all NES SRAM memory locations. Another is to maintain a list of SRAM locations which are being modified, much like the scheme used in load/store units of today's aggressive superscalar microprocessors. The size of NES SRAM memory, 32kB, makes keeping full-empty bits relatively expensive. The other design is also difficult because the exact SRAM memory location read or written by a command is sometimes unknown until part way through its execution in a functional unit.

Barrier Command: A middle of the road option with moderate design complexity is to have sP software explicitly specify dependences. The sP could be provided with "barrier" commands which block issue of subsequent commands until previous ones have completed. Given that "barriers" are expected to be used quite frequently, it can be made an option in each ordinary command. Setting this option bit is equivalent to preceding a command with a barrier command.

Dependence Bit Register File: Another way for sP software to explicitly specify dependence is to provide a small register file of full-empty bits. Each command is expanded with two or more fields for naming full-empty registers, one for which it is producer and the others consumer. A command is not issued until its consumer full-empty registers are all set. Its producer full-empty register is cleared when the command is issued, and set upon its completion. This approach provides finer granularity specification of dependence than the "barrier" command approach, but comes at the price of increased command size, dependence register file hardware, and the control logic to track and enforce dependence.

Multiple Sequentially Processed Queues: At the other extreme is a very simple design which executes only one command from each queue at a time, proceeding to the next one only after the previous one has completed. While simple, this design kills all inter-command parallelism. To improve parallelism, a design can employ multiple command queues, with no ordering constraints between commands in different queues. It can also limit ordering to command types that are likely to have dependences. In the unlikely event that ordering is needed between other commands, the sP will manually enforce it by appropriately holding back command issue.

The last design is adopted in the StarT-Voyager NES, where two command queues are supported. The sP can also utilize additional queues which are limited to sending messages only. All the design options described above achieve the first order requirement of allowing the sP to issue a stream of commands without polling for completion. They differ only in their exploitation of inter-command parallelism. Our choice is a compromise between design simplicity and parallelism exploitation.

4.4.2.2 Command Completion Notification

Command completion is reported to the sP via a completion queue. Since it is expensive for the sP to poll the NES Core, completion notification is made an option in each command. Very often, a sequence of commands requires completion notification only for the last command, or none at all. The completion queue is treated like another message queue, and can be one of the queues named in a OnePoll. Using OnePoll further reduces sP polling cost.

The completion queue has a finite size, and when it is full, completion notification cannot be placed into it. This will cause functional units to stall. The sP is responsible to ensure that this does not lead to a deadlock. It should, in general, pre-allocate space in the completion queue before issuing a command requiring completion notification. Alternatively, sP code can be structured to ensure that it is constantly removing notifications from the completion queue.

4.4.3 Option 3: Template Augmented Command and Completion Queues

Given that the sP is expected to issue short command sequences when processing new requests, it is natural to consider extending our design with *command templates*. The goal is to reduce the time taken to issue a command sequence by pre-programming the fixed portions of each command sequence in the NES Core. To invoke a sequence, the

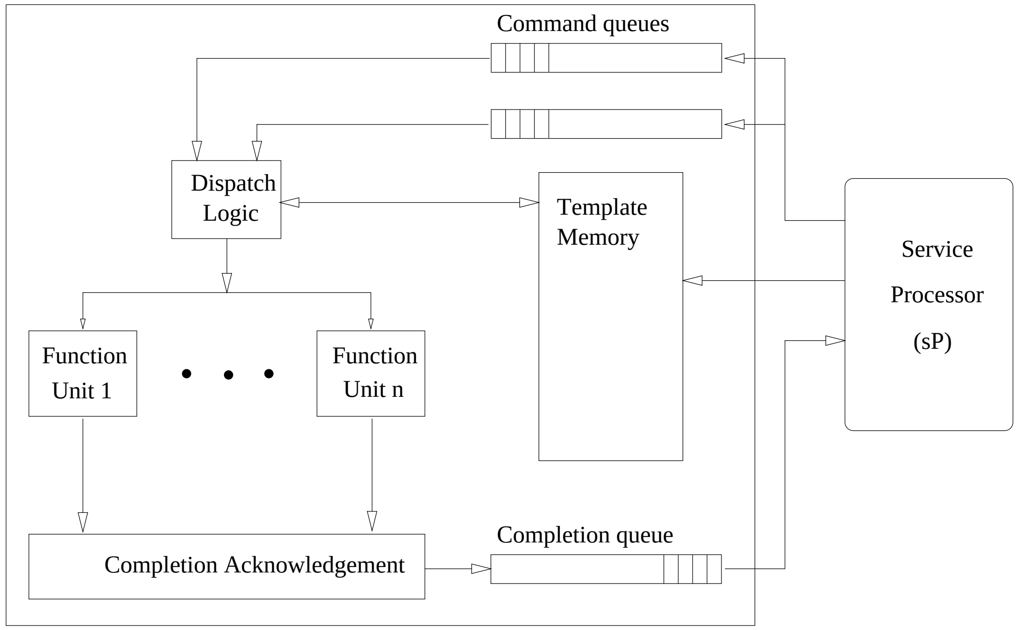


Figure 4-7: A template augmented command and status register interface.

sP simply identifies the template, and specifies the values of the variable parameters.

While interesting, it is unclear how much savings this produces. As mentioned before, when a stream of commands is issued by the sP, processing of earlier commands overlap issuing of subsequent ones. Thus the benefit of the template scheme is probably more a matter of reducing sP occupancy rather than overall latency of getting the command sequence done. Coming up with an efficient means of passing parameters is also tricky.

As proposed here, a template is a straight sequence of commands. It is, however, not difficult to envision including more capabilities, such as predicated commands and conditional branches. If the NES Core capability is pushed in that direction, one will soon end up with a customized programmable core. At that point, the sP is probably no longer needed. A substantial amount of design and implementation effort is needed to implement the template scheme, and even more for a more generally programmable NES Core. Consequently, we did not explore the idea much further.

4.5 NES Core Micro-architecture

This section describes in detail the different queues and functional units in the NES Core as illustrated in Figure 4-8. We do this in an incremental fashion, starting with the simplest subset which implements only Resident Basic message support. By re-using and extending existing hardware capabilities, and adding additional NES Core hardware, the NES's functionality is enhanced until we arrive at the final design. Figure 4-9 shows the partitioning of the NES Core functional units and queues into physical devices.

4.5.1 Resident Basic Message

Figure 4-10 presents a logical view of the NES Core components that implement Resident Basic message passing support. The state associated with each queue is separated into two parts: the buffer space and the control state. Control state is located with the transmit and receive functional units, while buffer space is provided by normal dual-ported synchronous SRAM. The exact location and size of each queue's buffer space is programmable by setting appropriate base and bound registers in the queue's control state. To support a number of queues with minimal duplication of hardware, the message queue control state is aggregated into "files", similar to register files, which share control logic that choreographs the launch and arrival of messages to and from the network.

Software accesses the message queue buffer space directly using either cached or uncached bus transactions. The NES State Read/Write logic provides software with two windows to access message queue control state. One window has access to the full state including configuration information. Obviously, access to this window should be limited to system software or other trusted software. A second window provides limited access to only producer and consumer pointers. This is exported to user-level code.

The actual implementation of Basic Message functions involves the devices shown in Figure 4-11. The design is completely symmetrical for the aP and the sP. Two

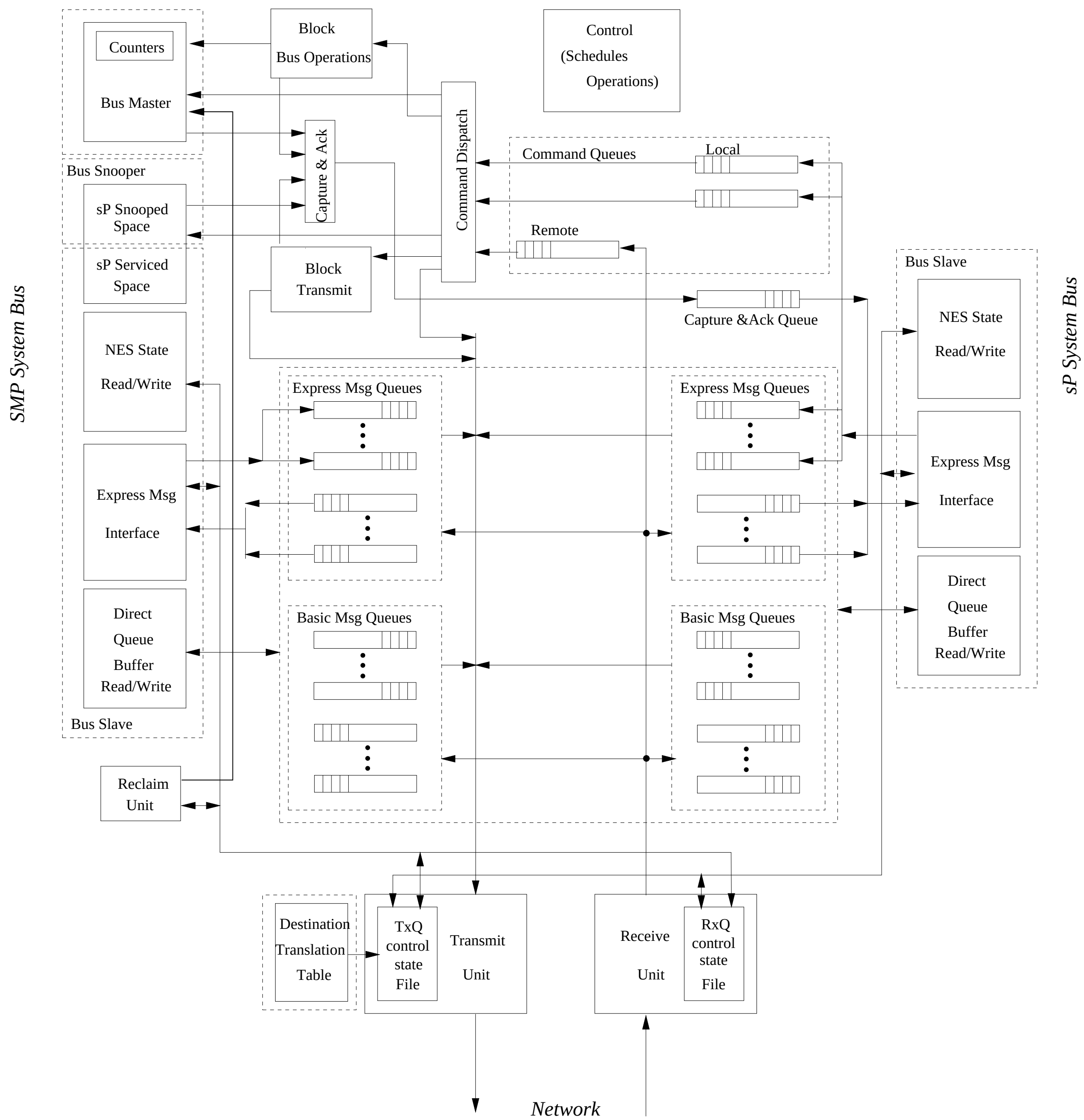


Figure 4-8: Queues and functional units in the StarT-Voyager NES.

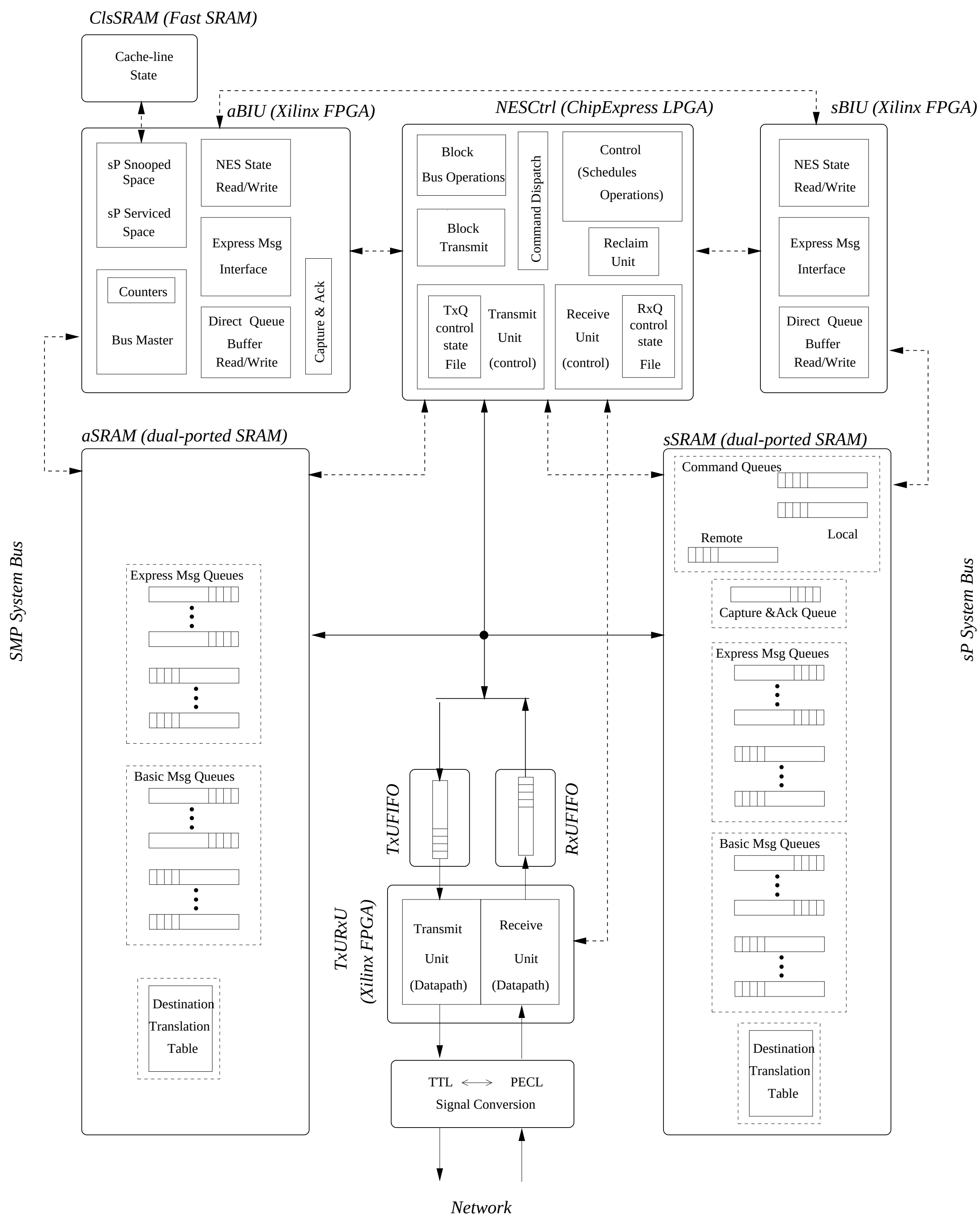


Figure 4-9: Physical devices in the StarT-Voyager NES.

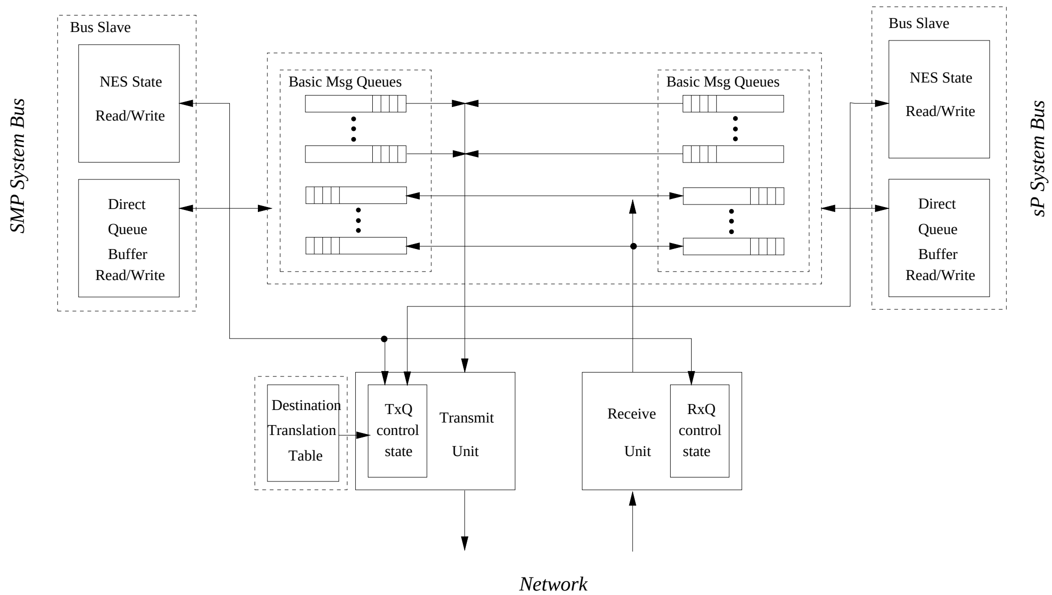


Figure 4-10: Queues and functional units to implement Resident Basic Message.

banks of dual-ported SRAM provide storage space for message buffer. Software access to these SRAM banks is achieved with the help of the BIU's, which determine the SRAM location to read data from and write data into. The destination translation tables are also located in the SRAM banks. The TxFIFO and RxFIFO in the diagram are used to decouple the real-time requirement of Arctic network from the scheduling of I-bus. This is necessary for in-coming messages to prevent data losses. A second role of these FIFO's is for crossing clock domains: the network operates at 37.5 MHz, most of the NES at 35MHz.

Logic that implements transmit and receive functions is divided into two parts: one part resides in the NESCtrl chip, and the other in the TxURxU FPGA. The former is responsible for the control functions. Although it observes packet headers, this part does not manipulate nor alter the data stream. Such tasks are the responsibility of the TxURxU. The NESCtrl chip is also the arbitrator for the I-bus, which is shared among many different functions.

Access to Queue Control State

Implementation of control state update is easy, since both the name of the state

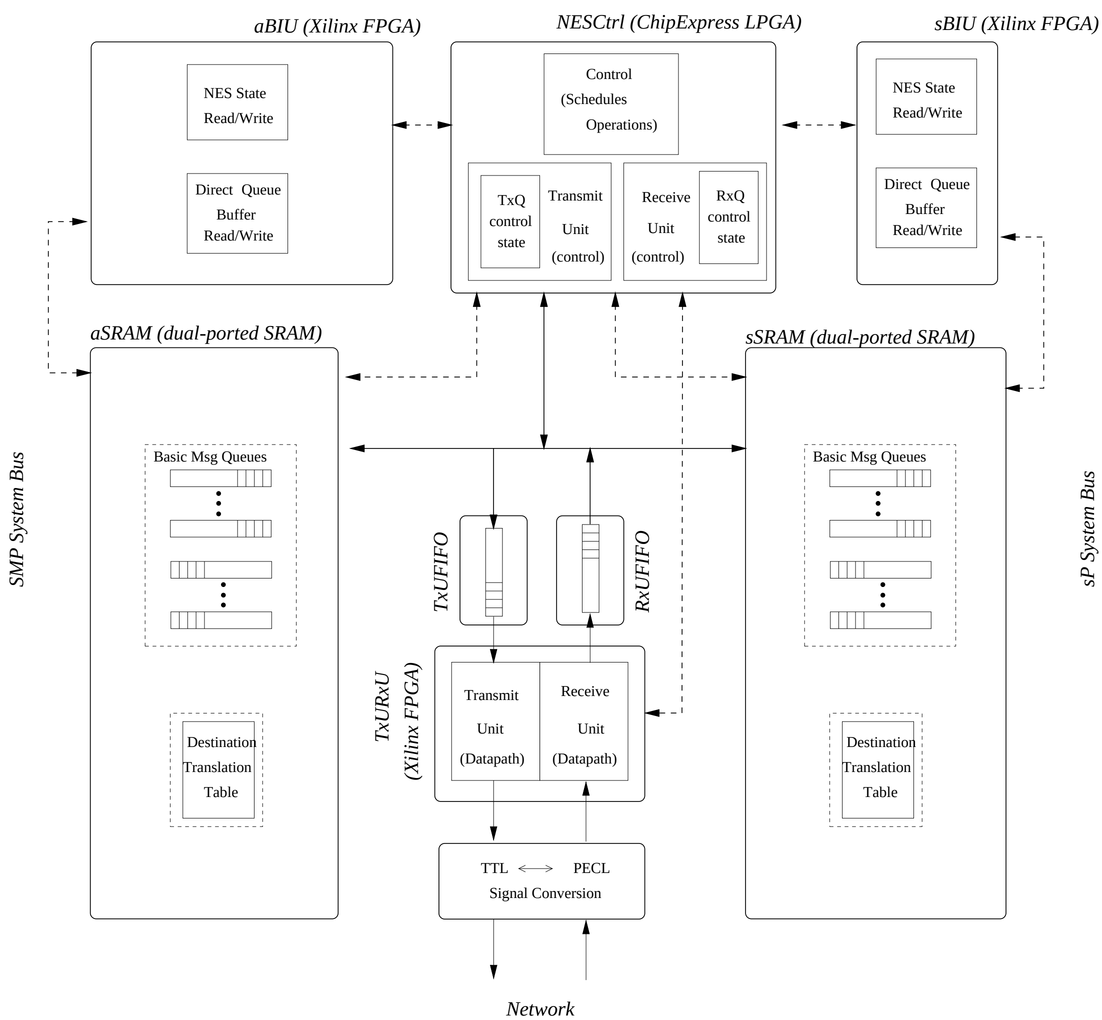


Figure 4-11: Actual realization of Resident Basic Message queues and functional units.

being updated, and the new value are both found in the *address* portion of a control state update bus transaction. Reading control state is more involved, because the data has to be written from the NESCtrl chip over the I-bus into the SRAM banks before it can be supplied as data to read bus transactions. To reduce the impact of this long data path on the latency of reading queue state, shadow copies of the most frequently read state, the producer pointer of receive queues and the consumer pointer of transmit queues, are kept in the SRAM banks, and periodically updated by NESCtrl.

Transmit Unit

The transmit unit tracks the transmit queues that are non-empty, updating this information each time the producer or consumer pointer of a transmit queue is written to. A scheduler in the transmit unit then selects one of the non-empty transmit queues that is also enabled and loads its control state into the state machine that controls the actual message launch.

The transmit unit is responsible for performing destination translation on most out-going packets. With both message and translation table data coming from NES SRAM, this is achieved by simply marshalling the appropriate translation table entry data into the message data stream heading towards the TxURxU. The data path in TxURxU then splices that information into appropriate portions of the out-going packet's header.

For the other packets that request physical destination addressing, the transmit unit checks that the transmit queue from which the packet originates has physical addressing privilege. Any error, such as naming a logical destination whose translation table is invalid, or using physical addressing in a queue without that privilege results in an error that shuts down the transmit queue. The error is reported in an NES Error register, and may raise either an aP or an sP interrupt depending on how the aP and sP interrupt masks are configured.

Receive Unit

The receive unit is responsible for directing an incoming packet into an appropriate receive queue. It maintains an associative lookup table which matches RQID (Receive Queue ID), taken from the header of incoming packets, to physical message queues. The receive unit also maintains a special queue called the Overflow/Miss queue. Packets with RQID's that miss in the lookup are placed into this queue. So are packets heading to receive queues that are already full, and whose control state enables overflow. Exceptions to the latter are packets that ask to be dropped when its receive queue is full. The choice of whether to drop a packet heading to a full receive queue is made at the packet source, specified in the translation table entry.

A packet can also block if its receive queue is full. This happens when the receive queue state indicates that overflow is not enabled, and the incoming packet does not ask to be dropped when encountering such a situation. Although blocking is in general dangerous, providing this option in the hardware gives system software the flexibility to decide whether this should be made available to any software; perhaps sP software can make use of it.

Link-level Protocol

Both the transmit and receive units participate in link-level flow-control with an Arctic Router switch. Arctic adopts the strategy that each out-going port keeps a count of the number of message buffers available at its destination. This is decremented each time a packet is sent out of a port, and incremented when the destination signals that space for another packet is available. Arctic also imposes the rule that the last packet buffer has to be reserved for high priority packets. The NES's transmit unit has to respect this convention. The receive unit typically indicates a packet buffer space has freed up when a packet is moved from RxFIFO into message buffer space in SRAM.

4.5.2 Resident Express Message

The NES Core implementation of Resident Express message support, shown in Figure 4-12, re-uses most of the hardware that is present for Basic message. The hardware message queue structures are re-used with only minor changes to the granularity

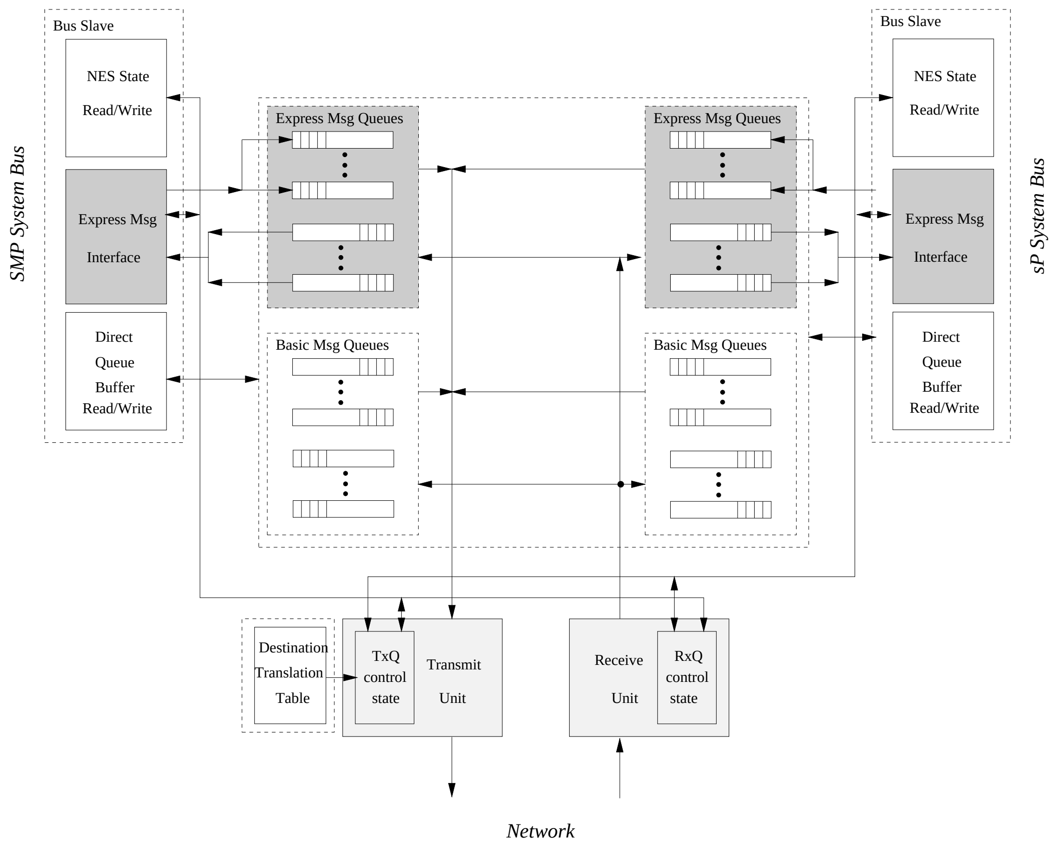


Figure 4-12: Queues and functional units to implement Resident Express message and Basic message. The lightly shaded portions require modifications when we add Express message. The more deeply shaded portions are new additions for Express message.

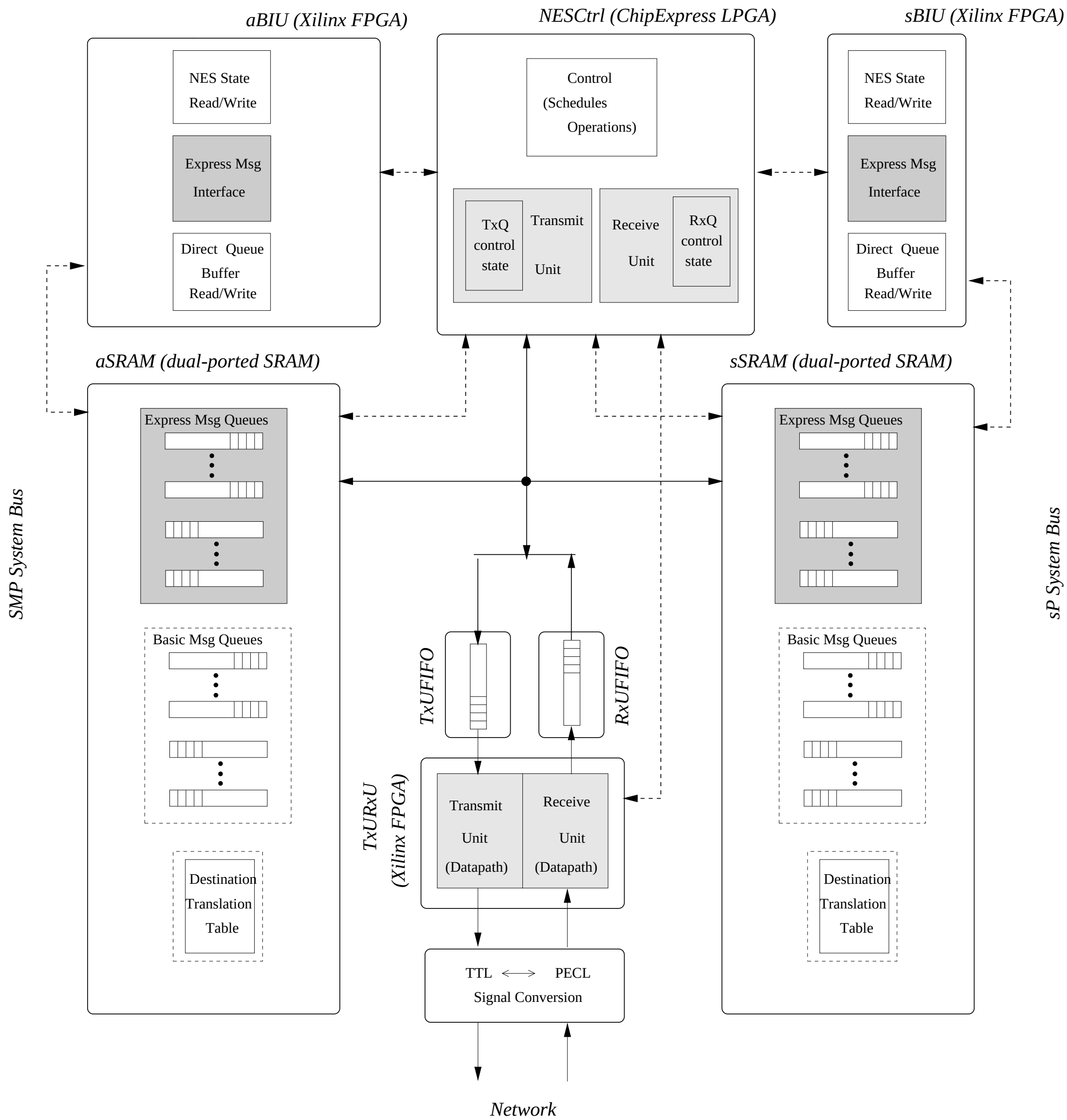


Figure 4-13: Actual realization of Express and Basic Message queues and functional units. The lightly shaded portions require modifications when we add Express message. The more deeply shaded portions are new additions for Express message.

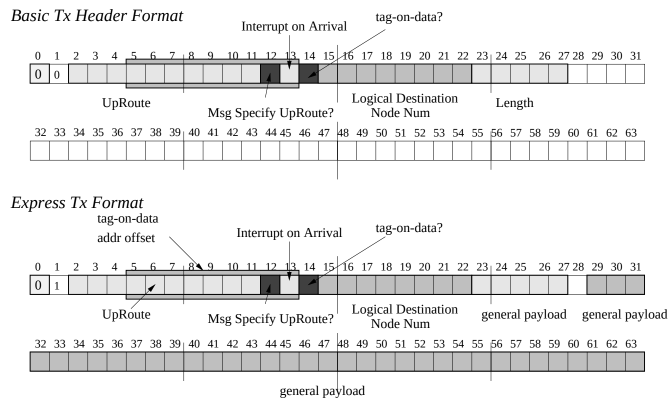


Figure 4-14: Message headers of Basic and Express messages.

of the pointers, 64 bits vs 32 bytes in Basic message.

Transmit Unit

The Basic message and Express message formats are compared in Figure 4-14. Because we keep the first 4 bytes of the Basic and Express message headers almost identical, few changes are needed in the transmit unit. One change is having the transmit unit add a packet length field, which is explicitly specified by software in Basic message format, but left implicit in the Express message format. Another change is feeding the 64 bit Express message to TxURxU twice, the first time as the equivalent of the Basic message header, and the second time as the equivalent of the first data word in Basic message.

Receive Unit

The receive unit requires additions to re-format Arctic packets into the 64 bit format required by Express message. No re-formatting is needed for Basic message since its receive packet format is essentially Arctic's format packet. The re-formating, shown in Figure 4-15, is very simple, and is done in TxURxU. Most existing functions of the receive unit, such as respecting Arctic's link-level protocol and determining the NES SRAM location to place an incoming packet, are unchanged between Basic and Express message.

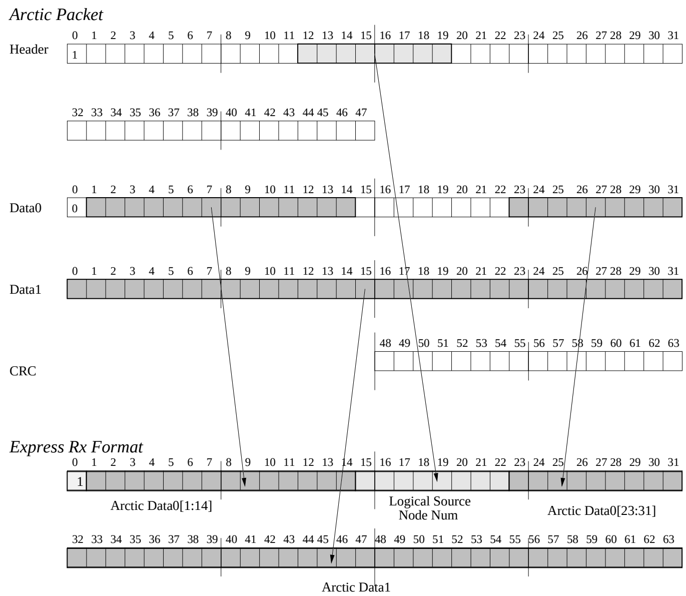


Figure 4-15: Transforming Arctic packet into Express messages receive format.

BIU Express Message Support

The main hardware addition to support Express message is in the BIU's, where shadow copies of the control state of Express message queue are introduced. The BIU's use these to implement the hardware FIFO interface of Express messages, and also the OnePoll mechanism for prioritized polling of message queues.

When processor software performs an Express message send with an uncached write, its BIU uses the local copy of queue state to generate the message buffer's SRAM address. The BIU also writes the address portion of the bus transaction, which is part of the message, into the even word of this buffer if this transmit is triggered by a 4-byte uncached write. This is done over the I-bus, with the help of NESCtrl. Finally, it increments its copy of the producer pointer, and updates the transmit unit's copy of that pointer. The latter is done using the same hardware path taken when queue state is explicitly updated by software in a Basic message send.

When software polls for messages from an Express message queue, the BIU again uses its copy of queue state to generate the SRAM address. But instead of always generating the SRAM address of the buffer indicated by the queue's consumer pointer, the BIU takes into account whether the queue is empty. If it is empty, the BIU generates the SRAM address of a special location that contains a system software programmed Empty Express message. If the queue is non-empty, the BIU updates the consumer pointer. The BIU and the receive unit co-operate to keep the queue pointers in synchrony. The BIU propagates consumer pointer updates to the receive unit, while the receive unit propagates producer pointer updates in the reverse direction.

4.5.3 OnePoll

OnePoll of multiple queues is an extension of polling from a single Express message receive queue. A OnePoll bus transaction specifies a number of message queues that it wants to poll from. On its part, the BIU determines which queues among these are not empty, and when there are several, one queue is selected based on a fixed, hardwired priority. Since normal polling already selects between two possible SRAM addresses to read data from, OnePoll simply increases the choice to include more queues.

To reduce the amount of work done when a OnePoll transaction is encountered, the BIU maintains a 1-bit value for each Express message receive queue indicating whether it is empty. This is re-evaluated each time a producer or consumer pointer is updated[8]. When servicing a OnePoll bus transaction, the BIU uses this information, selecting only those of queues picked by the OnePoll bus transaction, and finds the highest priority non-empty queue among them. The result is then used as mux controls to select the desired SRAM address. The depth of OnePoll logic for n queues is $O(log(n))$, while the size of the logic is $O(n^2)$. In our implementation, we are able to include up to ten queues without timing problem.

A Basic message receive queue can also be included as a target queue in a OnePoll operation. When a Basic message queue is the highest priority non-empty queue, the data returned by the OnePoll action includes both a software programmable portion

[8]This implementation also avoids the need for parallel access to a large number of producer and consumer pointers. These pointers can continue to be kept in register files with limited number of read/write ports.

that is typically programmed to identify the Basic message queue, and a portion containing the NESCtrl updated shadow queue pointers in NES SRAM.

4.5.4 TagOn Capability

TagOn capability is an option available with both Basic and Express message types. When transmitting a TagOn message, the transmit unit appends additional data from an NES SRAM location specified in the message. When receiving an Express TagOn message, the receive unit splits the packet into two parts, directing the first part into a normal Express receive queue, and the second into a queue similar to Basic message receive queue. Modifications to the control logic of both the transmit and receive units are necessary to implement this feature.

Implementation of TagOn also requires additional control state. Transmit queues are augmented with a TagOn base address, and a bound value. With the use of these base/bound values, the SRAM address of the additional data is specified with only an offset, reducing the number of address bits needed. This is helpful in Express TagOn where bits are scarce. The scheme also enforces protection by limiting the NES SRAM region that software can specify as the source of TagOn data. The control state of Express receive queue is increased to accommodate the additional queue.

When performing TagOn transmit, the transmit unit needs to read in the TagOn address offset to generate the TagOn data's SRAM address. This capability is already present to generate addresses of translation table entries. Thus the design changes are fairly incremental.

4.5.5 NES Reclaim

The NES provides the Reclaim option[9] to help maintain coherence of cache-able Basic message queue buffer space. This is implemented in a very simple fashion. Instead of actually maintaining coherence state-bits for each cache-line of buffer space,

[9] Reclaim option is only available to the aP. Supporting Reclaim requires the BIU to be a bus master. Because there is no pressing need for sBIU to be a bus master, we did not go through the effort of implementing it.

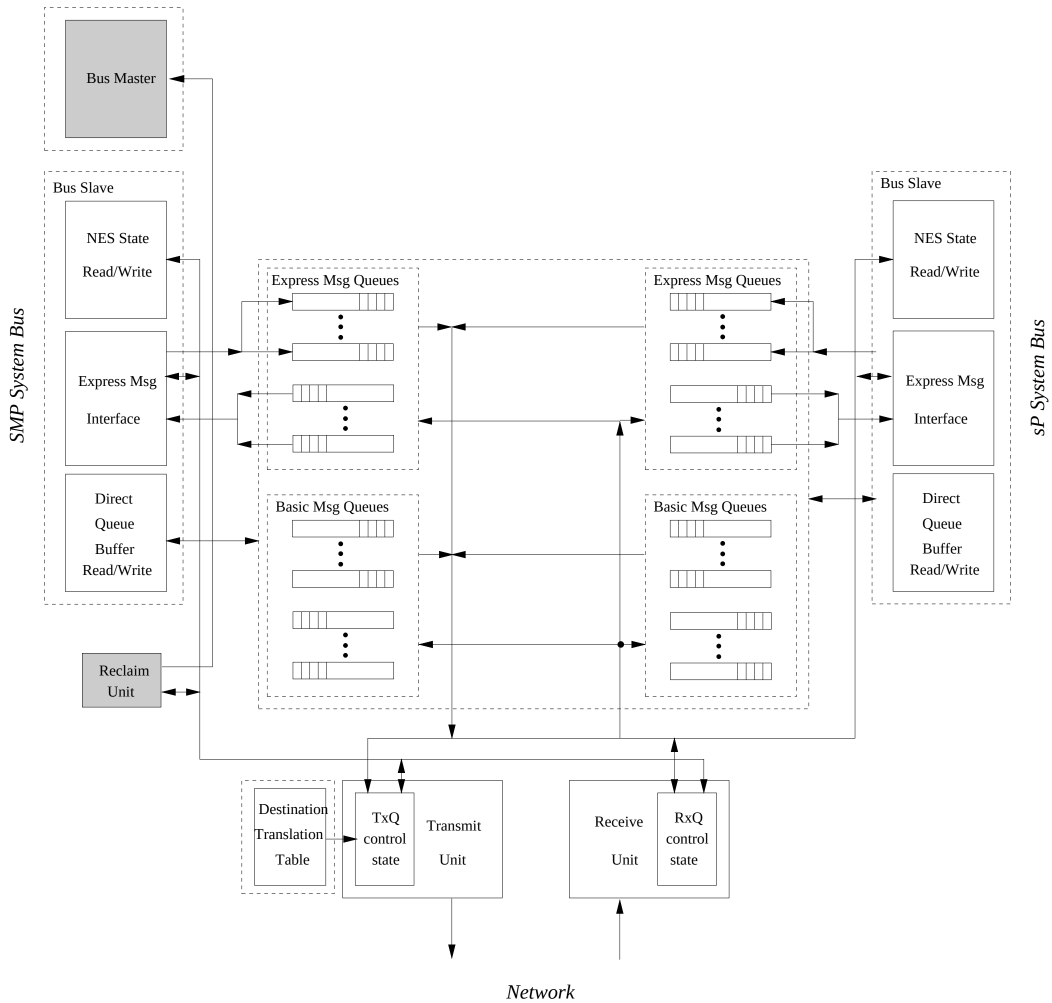


Figure 4-16: Queues and functional units to implement Resident Express message and Basic message, including Reclaim option for Basic message queues. The shaded portions are introduced to support Reclaim.

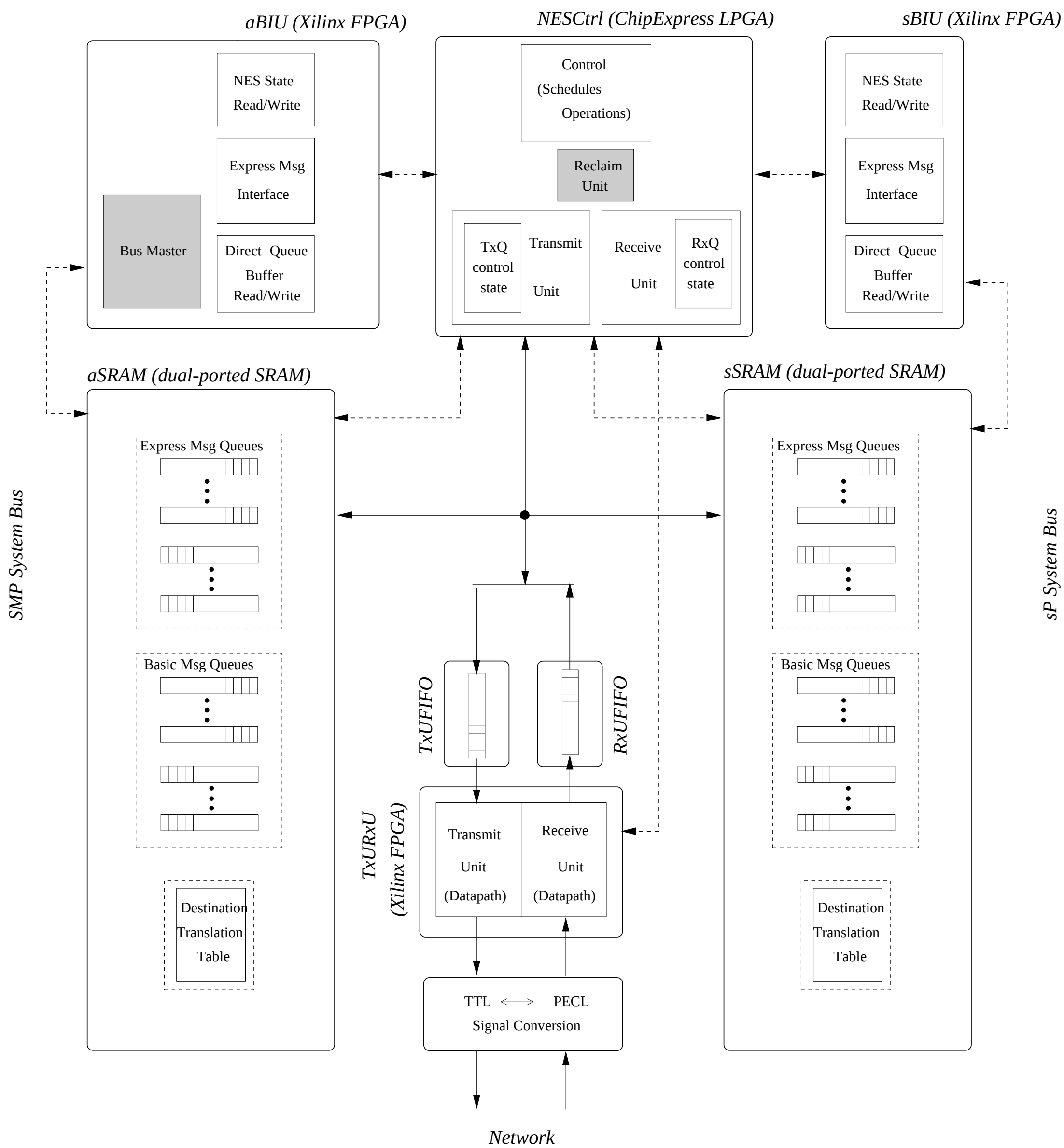


Figure 4-17: Actual realization of Express and Basic Message queues and functional units, including Reclaim option for Basic message queues. The shaded portions are introduced to support Reclaim.

the design relies on queue pointer updates to trigger coherence maintenance actions. When the producer pointer of a transmit queue is updated, the region between its old value and the new value is conservatively assumed to be freshly written, and still in the processor's cache. The NES triggers write-back of these cache-lines using appropriate system bus transactions before transmitting the messages.

Similarly, when the consumer pointer of a receive queue is updated, the region between its old and new values is conservatively assumed to be in the processor's cache. This time, the value needs to be invalidated as they will become stale once the buffer space is re-used for new messages.

As shown in Figures 4-16 and 4-17, the NES Core implements Reclaim with the addition of a *Bus Master Unit* to the aBIU, and a new *Reclaim Unit* to the NESCtrl. The Reclaim Unit keeps a new category of pointers, called reclaim pointers, that are related to producer and consumer pointers. We will use transmit queue operations to illustrate how this scheme works.

During a message send, software updates the reclaim pointer instead of the producer pointer of a transmit queue. When the Reclaim Unit detects that a reclaim pointer has advanced beyond its corresponding producer pointer, the Reclaim Unit issues bus transaction requests to the BIU's Bus Master unit to pull any dirty data out of processor caches. Once these bus transactions complete, the Reclaim Unit updates the queue's producer pointer, which triggers the transmit unit to begin processing. With this implementation, neither the transmit nor receive units are modified to support reclaim. Furthermore, the option of whether to use reclaim simply depends on which pointer software updates.

4.5.6 sP Bus Master Capability on SMP System Bus

With the NES Core functions described so far, the sP is unable to do much as an embedded processor. In fact, an sP receives no more services from the NES Core than an aP. But now that aBIU has implemented bus master capability, it is only a small step to make this capability available to the sP. Figure 4-18 shows the additional functional blocks and queues that are needed to make this possible. Figure 4-19 is

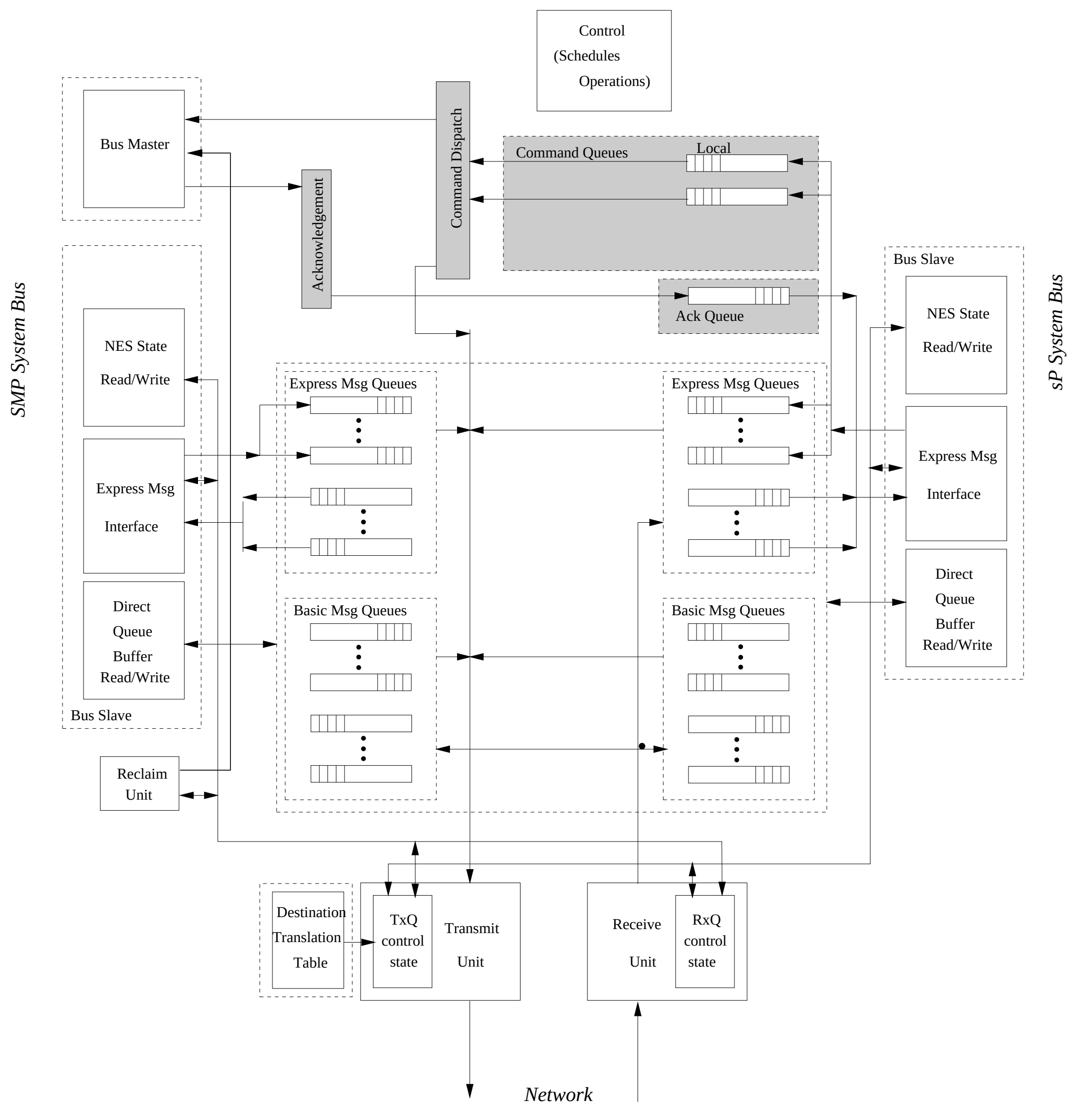


Figure 4-18: Queues and functional units to implement Resident Express message, Basic message (including Reclaim), and giving sP the ability to initiate bus transactions on the SMP system bus. The latter is made possible with the addition of the shaded regions.

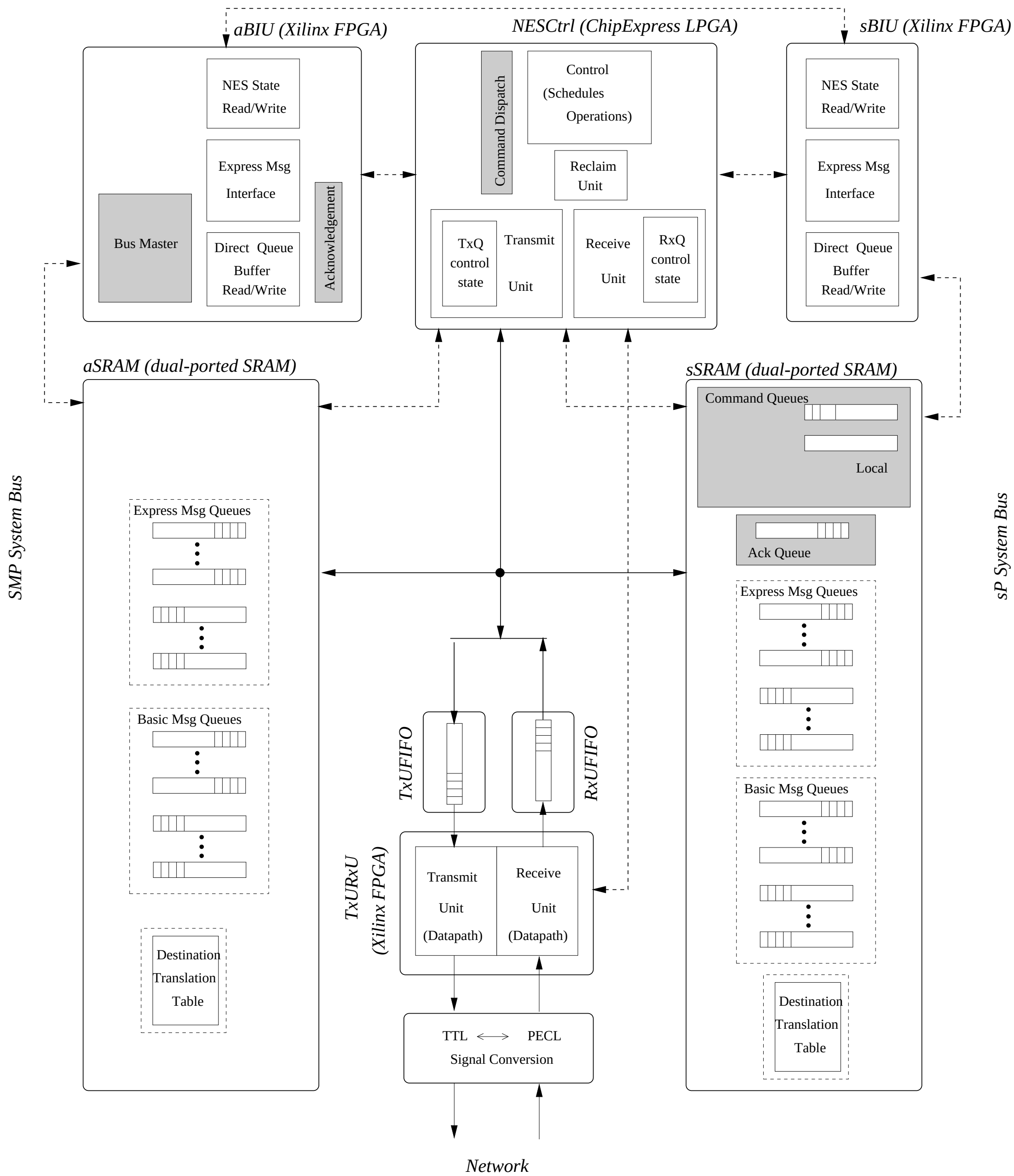


Figure 4-19: Actual realization of Resident Express and Basic message functions (including Reclaim), and giving sP the ability to initiate bus transactions on the SMP system bus. The latter is made possible with the addition of the shaded regions.

a similar diagram showing how these components map onto physical devices. The main additions are two *Local Command Queues*, a *Dispatch Unit* that arbitrates and dispatches commands from these queues, and an *Acknowledgement Queue*.

Local Command Queues To execute bus operations on the SMP system bus, the sP inserts bus operation commands into any of the Local Command queues, in the same way it issues an Express message send. One can think of this as issuing a message to the bus master unit in the aBIU to perform the bus operation. A bus operation command specifies the transaction's (physical) address and control signals. In addition, if data is involved in the transaction, the command also specifies the aSRAM address to read data from or write data to. The format of this bus operation request message is also used by the Reclaim Unit to request bus operations. In this way, the Bus Master unit offers a primitive that is used by both the Reclaim Unit and the sP.

Acknowledgement Queue If the sP needs to know when a command is completed, it can request for an acknowledgement. The Bus Master unit inserts such an acknowledgement into an Acknowledgement queue using much of the infrastructure that already exists to place the address portion of an Express message into buffer space in the SRAM banks. A minor difference is that an acknowledgement requires 64 bits of information to be written as opposed to 32 bits in the case of Express message composition. The sP polls for acknowledgement in the same way it receives Express messages. In fact, using OnePoll, the sP can be polling from the Acknowledgement queue and other message queues simultaneously.

The above description shows that most of the functions needed to let the sP issue bus transactions on the SMP system bus already exist. Giving sP the ability to transfer data between the SMP system bus and the aSRAM, which it can access directly using load/store instructions, gives the sP indirect access to the host SMP's main memory. Furthermore, data can be transferred between the SMP main memory and NES SRAM locations used as message buffer space. Thus, the sP can now be a proxy that transmits messages out of aP DRAM, and writes in-coming messages into aP DRAM.

The sP can also send normal Express and Express TagOn messages from the Local Command queues. Because bus operation and message commands in these queues are serviced in strict sequential order, the sP can issue both a bus command to move data into aSRAM and an Express TagOn command to ship it out all at once; there is no need for the sP to poll for the completion of the bus command before issuing the Express TagOn.

While this level of NES support is functionally adequate for the sP to implement Non-resident Basic message queues, other NES Core features described in Sections 4.5.8 and 4.5.9 improve efficiency.

4.5.7 Inter-node DMA

With the NES Core features described so far, the sP can implement DMA in firmware. But such an implementation can consume a significant amount of sP time (see Section 5.4). Since block DMA involves a number of simple but repetitive steps, we added several functional units in the NES Core to off-load these tasks from the sP.

Two functional units, the *Block Bus Operations unit* and the *Block Transmit unit*, are employed at the sender NES. The receiver NES is assisted by the addition of a *Remote Command Queue*, and a *counting service* provided by the Bus Master unit. The basic idea is to transmit DMA packets that include both data, *and bus commands*. When the latter are executed by the Bus Master unit at the destination NES, data is written to appropriate SMP main memory locations. The counting service counts the number of packets that have arrived so that when all packets of a transfer have arrived, an acknowledgement is inserted into the Acknowledgement queue. The sP is responsible for setting up the Block Bus Operations unit, the Block Transmit unit, and initializing the DMA Channel Counters. It does this with commands issued to the Local Command queues. All these commands include acknowledgement options. Details of these features follow.

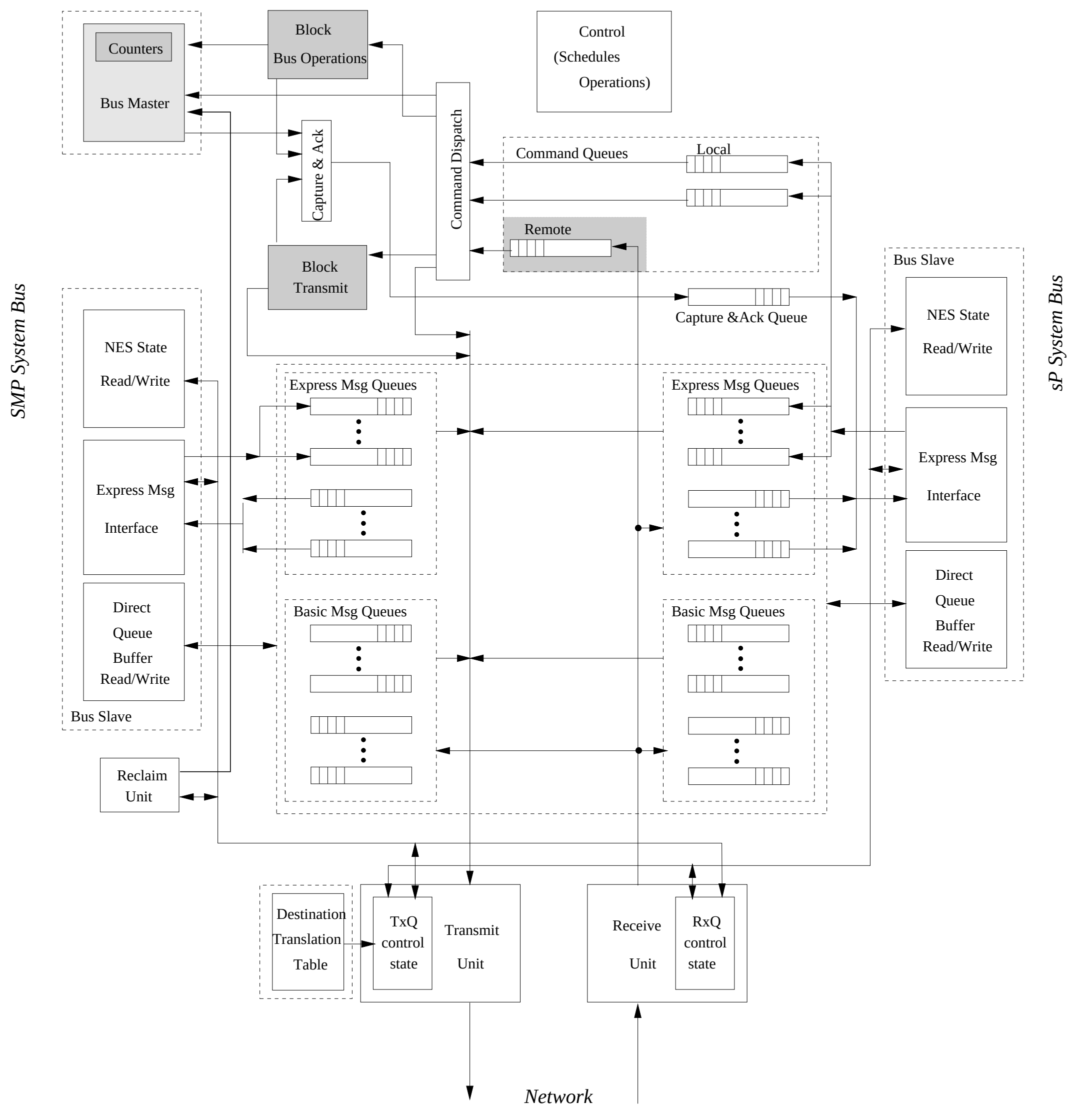


Figure 4-20: This diagram illustrates the addition of DMA support. The lightly shaded blocks are modified while the darkly shaded regions are new additions to provide NES hardware DMA support.

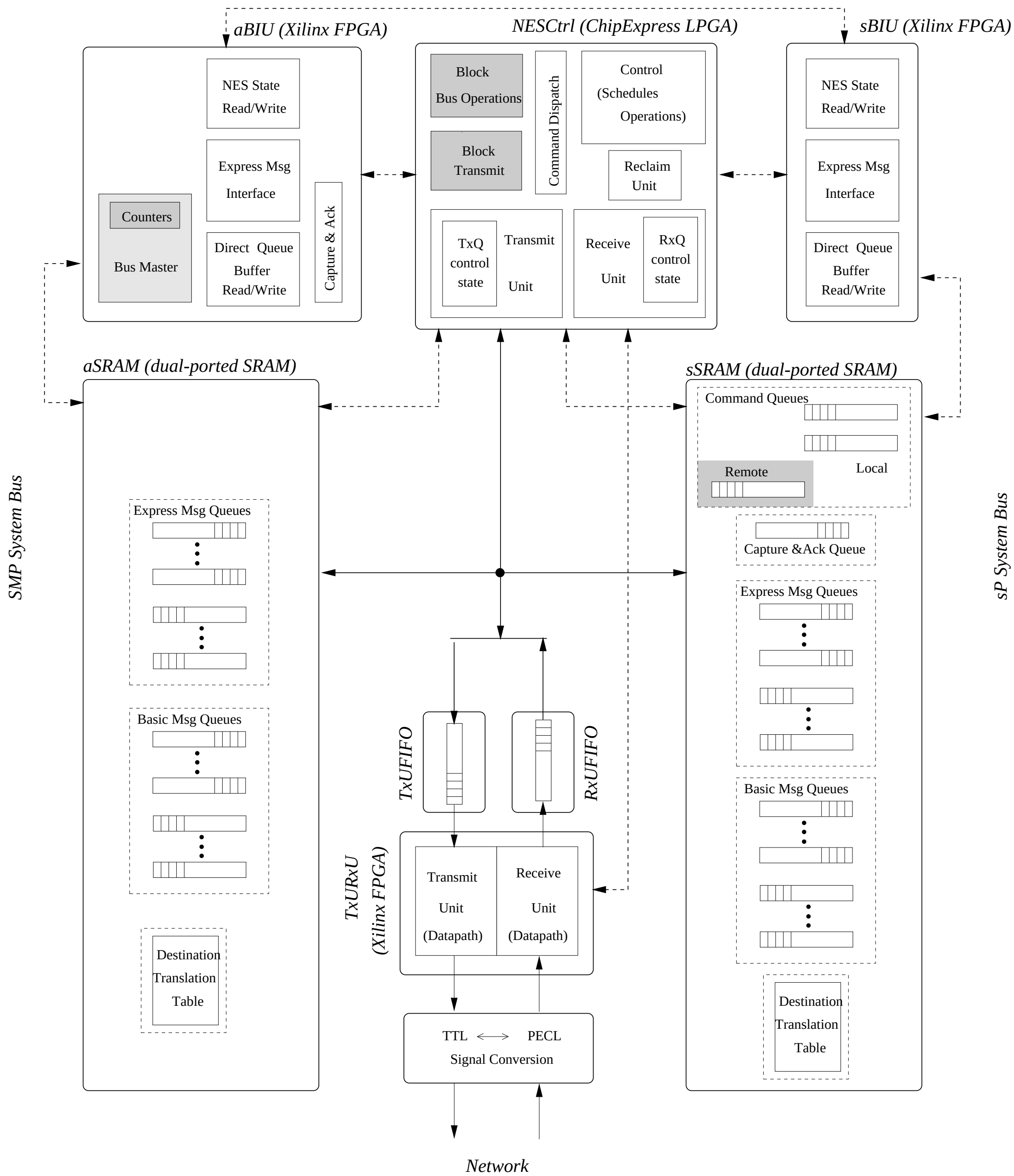


Figure 4-21: Actual realization of newly added logic for DMA support are shaded in this diagram. Lightly shaded regions are existing functional blocks that are modified, while darkly shaded regions are new functional blocks.

Block Bus Operations Unit: The Block Bus Operations unit is used to repeat a bus operation command a number of times, each time incrementing the SMP bus address and SRAM address by 32 (cache-line size)[10].

A command to this unit is similar to a bus operation command but also specifies the number of times the bus operation is repeated. The Block Bus Operation unit treats SRAM space very much like a message queue. Configurable system state in the Block Bus Operation unit includes base and bound addresses and a producer pointer that are used to generate aSRAM addresses. The base and bound values are not expected to change frequently so they are programmed through the usual NES state access mechanism and not via the command queues.

In order for the Block Bus Operations unit to operate in a pipelined fashion with the Block Transmit unit, the producer pointer is shared with the Block Transmit unit. It can be optionally reset in each command.

The Block Bus Operation unit is implemented in NESCtrl. It could equally well have been associated with the Bus Master unit in the aBIU FPGA.

Block Transmit Unit: The Block Transmit unit formats data into DMA packets, appending an appropriate header and two bus operation commands as trailers. Two cache-lines (64 bytes) of data, taken from NES SRAM, is sandwiched between them. A command to this unit specifies the physical destination node name, details of the bus operation command to use, such as the transaction type and other control signal values and the starting SMP system bus physical address. The command also specifies a packet count.

This unit shares the base, bound, and producer pointer state with the Block Bus Operations unit. It also uses a private consumer pointer. As long as the

[10] The design will be much more flexible if the increment is not fixed but can be varied in each command. Furthermore, there should be two separate increment values: one for system bus address, the other for SRAM address. Such a design would allow the Block Bus Operations unit to "gather" data from the SMP system bus into NES SRAM, subject to the data path-imposed constraints that the smallest data granularity is 8 bytes, and addresses have to be 8-byte aligned.

desired packet count has not been reached, and the producer pointer is ahead of the consumer pointer, this unit generates DMA packets. Data is fetched from aSRAM addresses produced from the base, bound and incrementing consumer pointer. The generated bus operation commands carry remote system bus addresses in increments of 32 (cache-line size)[11].

Each command to this unit can optionally reset the consumer pointer. This, together with the decoupling of the read operation from the transmit operation, allows multi-cast DMA operation at no additional design or hardware cost. A single block read command executed by the Block Bus Operations unit moves the desired data into NES SRAM, and multiple Block Transmit commands send them to several destinations.

The Block Transmit unit shares features with the normal Transmit unit and is implemented in NESCtrl.

Remote Command Queue: At its destination, a DMA packet is split into two parts, command and data, and enqueued into the Remote Command queue, a special receive queue. The command part of the Remote Command queue offers an instruction set that is almost identical to that of the Local Command queues. Thus, supporting a Remote Command queue adds few new requirements to what is already in the design.

Since it is more efficient to move data directly from the receive queue buffer space and space in this queue is shared between packets from multiple sources and dynamically allocated the bus commands in each DMA packet do not specify the SRAM address at which data is located. It is possible to adopt a design where the destination NES inserts the SRAM address into each bus command

[11]Just like in the case of Block Bus Operation unit, the Block Transmit unit would have been more flexible if the address increment is programmable, with separate SRAM address and SMP physical address increments. This would have allowed "scatter" operation. Scattering of data with granularity smaller than a cache-line is still rather inefficient, because each packet can only cause two bus transactions at the destination. The best way to get around it is to have a smarter functional unit at the receive end which produces the required number of bus transactions, a unit similar in function to the Block Bus Operations unit.

before they are passed on to the Bus Master unit. This is inconvenient in the StarT-Voyager NES because of the way data path and control are organized. TxURxU is the part of the data path where this can be done quite easily and efficiently, but unfortunately the SRAM address is not available at that point.

Instead, we augment the Bus Master unit with a copy of the Remote Command queue state and a modified form of bus operation commands called *DMA Bus Operation* commands. The aBIU generates SRAM address for these commands from the consumer pointer and associated base and bound values of the Remote Command queue. This choice also has the beneficial effect of freeing up some bits in the DMA Bus Operation command for specifying a DMA Channel number.

DMA Channel Counters: The Bus Master unit maintains eight DMA Channel counters. These are initialized by commands from the local or remote command queues, and decremented when a DMA Bus Operation command is completed. When a counter reaches zero, an acknowledgement is inserted into the Acknowledgement queue to inform the local sP of the event.

With these functional units, the sP is only minimally involved in DMA. The sP's may have to perform address translation if a request from user code uses logical memory and node addresses. But once that is done, the destination sP only needs to initialize a DMA channel counter and then poll for acknowledgement of that command. In parallel, the source sP issues the Block Bus Operation unit a command to read the desired data into the aSRAM. After it is informed that the destination sP has initialized the DMA Channel counter, the source sP issues a Block Transmit command.

A DMA Channel counter can also be re-initialized from the Remote command queue. This can reduce latency of a transfer, but requires the source sP to be pre-allocated a DMA Channel counter, and to also send all the DMA packets in FIFO order, behind the reset command packet[12].

[12]The design can be made more flexible. Instead of using a counter that is decremented, we can

Having control over the transaction type used by the Block Bus operation unit is useful. In the context of DMA, the possible options, READ, RWITM (Read with intent to modify), and RWNITC (Read with no intent to cache), have different effects on caches that have previously fetched the data that is being sent out. READ will leave only shared copies in caches, RWITM will leave no copies, while RWNITC will not modify the cache states, *i.e.* a cache can retain ownership of a cache-line.

The functional units introduced for DMA have additional uses. The Block Bus operation unit is obviously useful for moving block data from aSRAM to aP DRAM, *e.g.* when we swap message queues between Resident and Non-resident implementations. It can also be used to issue Flush bus operations to a page that is being paged out. (The current design shares base/bound state values between the Block Bus Operation and Block Transmit units. This couples their operations too tightly. Greater flexibility can be achieved if each has its own copy of this state. The producer pointer should also be duplicated and a command to the Block Bus Operation unit should specify whether to increment the Block Transmit unit's producer pointer.)

The DMA Channel counters can be decremented by a command that does not include bus transactions. This allows them to be used as counters that are decremented remotely via the Remote Command queue. Uses include implementing a barrier, or accumulating the acknowledgement count of cache-coherence protocol invalidations. Although the sP could do this counting, this hardware support reduces sP occupancy, and results in faster processing of the packets.

The Remote Command queue opens up many possibilities. An sP can now issue commands to a remote NES Core, without the involvement of the remote sP. This cuts down latency, and remote sP occupancy. On the down side, this opens up some protection concerns. As long as it is safe to assume that remote sP and system code

use two counters, a target value and a current count. An acknowledgement is only generated if the two counters match, and the target value is non-zero. If both counters are reset to zero after they match, this design avoids the requirement that the reset command has to arrive before the data packets. A variant, which keeps the target value unmodified after both counters match avoids the need to re-initialize the counter for repeated transfers of the same size. To accommodate both, a reset command should specify whether to clear the target value register. This part of the design is in the aBIU FPGA and can easily be modified to implement this new design.

is trusted, there is no safety violation because access to this queue is controlled by destination address translation mechanism. Hence, user packets cannot normally get into this queue. If remote sP and system code is not to be trusted, some protection checks will be needed at the destination. This is not in our current design.

DMA Implementation Alternatives

We now compare this design with two other DMA design options which were not adopted. One design is completely state-less at the receiver end, with each DMA packet acknowledged to the sender which tracks the status of the transfer. By adding a (programmable) limit on the number of unacknowledged DMA packets, this design can also easily incorporate sender imposed traffic control to avoid congestion. If the sender already knows the physical addresses to use at the destination, possibly because this has been set up in an earlier transfer and is still valid, this design has the advantage of not requiring setup. Its main disadvantage is the extra acknowledgement traffic, and hardware to implement the acknowledgement.

We take the position that congestion control should be addressed at a level where all message traffic is covered. As for the setup delay, our design can also avoid the setup round-trip latency if the destination physical address is known, and a channel has been pre-allocated, as described earlier.

Another DMA design utilizes a much more elaborate receiver in which both the bus transaction command and the full destination address are generated at the destination NES. The DMA packets still has to contain address offset information if multiple network paths are exploited to improve performance. Otherwise, the packets have to arrive in a pre-determined order, effectively restricting them to in-order delivery. Destination setup is of course necessary in this design. It has the advantage of avoiding the overhead of sending the bus command and address over the network repeatedly. For example, in our design, each DMA packet carries 64 bytes of data, 16 bytes of bus command, and 8 bytes of Arctic header. The 16 bytes of bus command is a fairly high overhead, which can be reduced to 1 or 2 bytes if this alternate design is adopted. This alternate design also gives better protection since SMP memory addresses are generated locally.

We settled on our design because it introduces very little additional mechanism for receiving DMA packets, but instead re-uses, with minor augmentation, existing bus master command capability. Implementing our design requires adding an external command queue, which is a more general mechanism than a specialized DMA receiver.

4.5.8 sP Serviced Space

Using the NES Core features described so far, the aP and sP can only communicate either with messages, or via memory locations in NES SRAM or the SMP's main memory. The sP Serviced Space support described in this section allows the sP to directly participate in aP bus transactions at a low hardware level. This capability, together with the features described above, is functionally sufficient for implementing CC-NUMA style cache-coherent distributed shared memory.

The sP Serviced Space is a physical address region on the SMP system bus that is mapped to the NES, *i.e.* the NES behaves like memory, taking on the responsibility of supplying or accepting data. What distinguishes this region from normal memory, such as NES SRAM, is that the sP handles bus transactions to this space, *i.e.* the sP decides what data to supply to reads, what to do with the data of writes, and when each bus transaction is allowed to complete. In that sense, this address space is "active" and not merely a static repository of state. The following NES Core mechanisms together implement the sP Serviced Space.

Transaction Capture

The transaction capture mechanism informs the sP about bus transactions to the sP Serviced space. For instance, if a bus transaction initiated by the aP writes data to this region, the sP needs to be informed and given the full details such as the address, transaction type, caching information, and the data itself. With this information, the sP can decide what to do, such as writing the data to main memory of a remote node.

We extend the Acknowledgement queue for this purpose. When capturing a transaction, the NES Core inserts its address and control information as a 64-bit entry

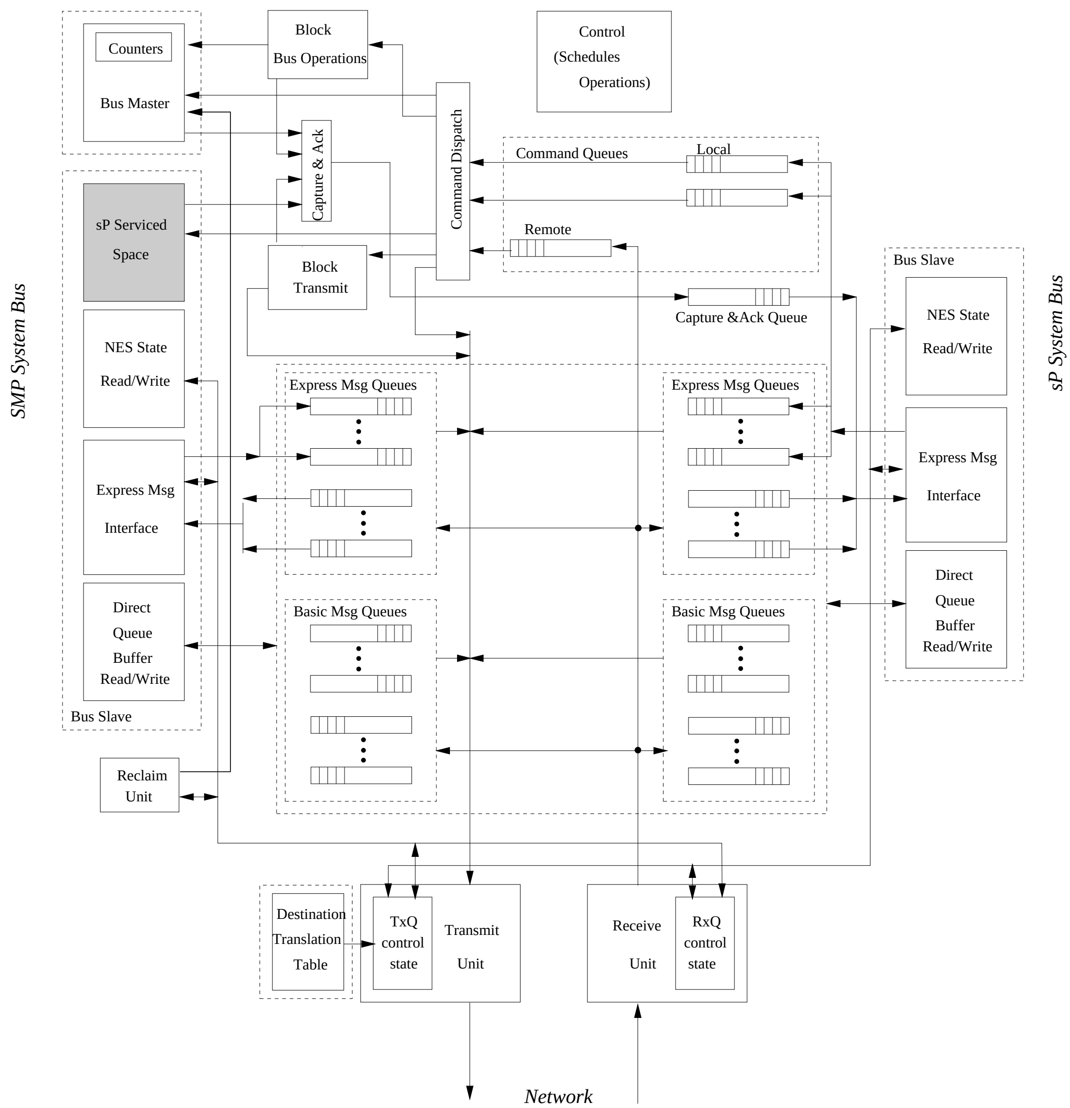


Figure 4-22: The addition of sP Serviced space support to the NES involves introducing the shaded functional block. Other existing infrastructure, such as the Capture & Ack queue, is re-used.

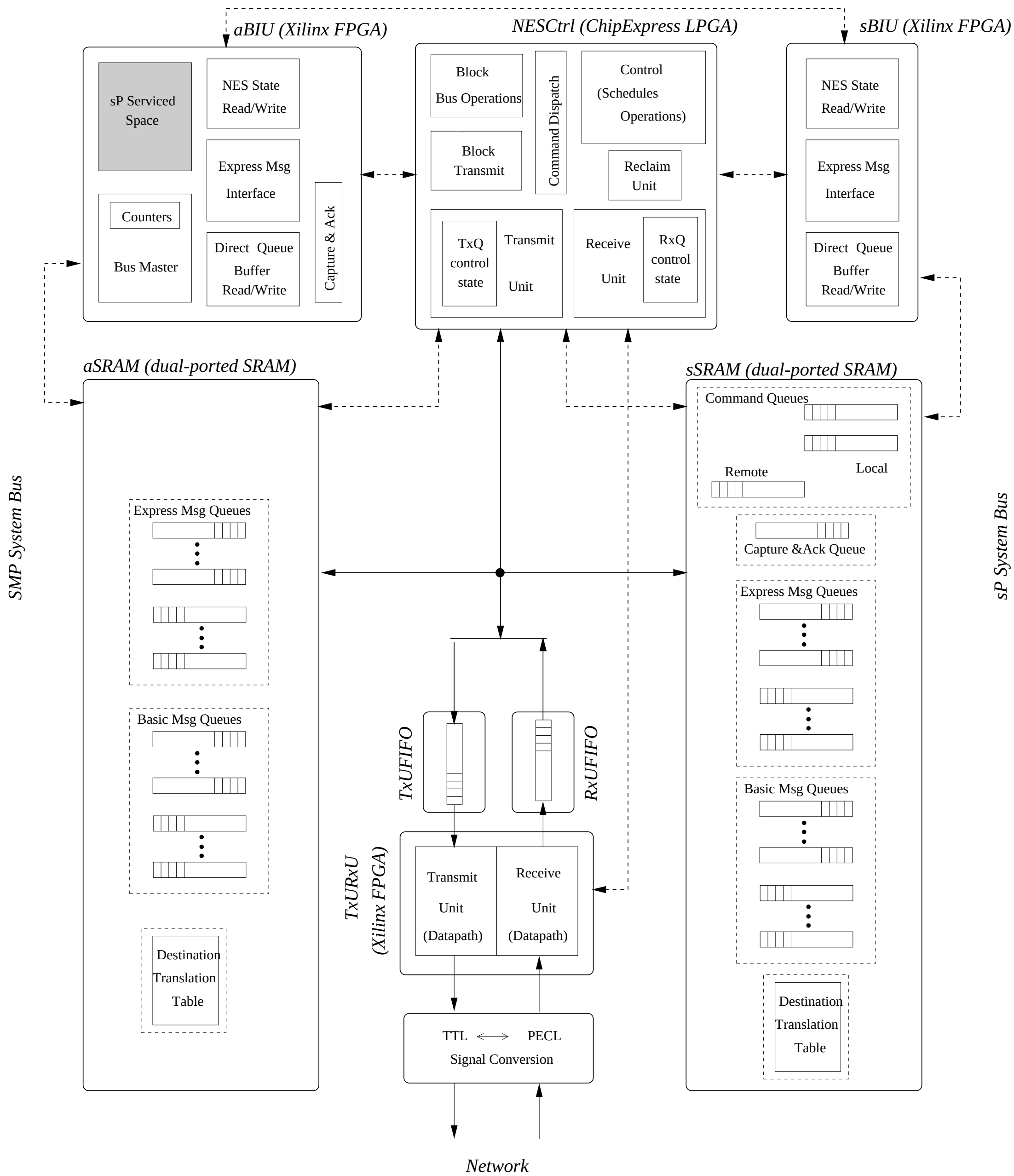


Figure 4-23: The addition of sP Serviced space support to the NES involves introducing the shaded functional block in the aBIU. Other existing infrastructure, such as the Capture & Ack queue, is re-used.

into the Acknowledgement queue, which we rename the *Capture & Ack* queue. As before, the sP polls this queue to obtain the captured information. In addition, a data queue is added to the Capture & Ack queue, into which data written by the bus transaction, if any, is enqueued. From an implementation point of view, the hardware data structure to implement this Acknowledgement queue is the same as that of an Express message receive queue, which has both a "header portion" that stores 64-bit entries, and a data portion. Thus, existing design and implementation component is re-used.

Transaction Approval

In order for the sP to return arbitrary data to a read-like bus transaction, the NES Core needs to "suspend" a bus transaction until the sP provides the data to return. To avoid deadlock situations such as those described in Section 3.4 when this address space is used to implement CC-NUMA shared memory, the means of suspending the bus transaction must not block the system bus. For the 60X bus protocol, this requires retrying the bus transaction, *i.e.* the bus master for the transaction is told to relinquish the bus, and re-attempt that transaction at a later time[13]. Since the bus master may re-attempt the same bus transaction several times before the sP is ready with the data, the NES Core should filter out repeated attempts; the sP should not be sent multiple copies of the bus transaction's address and control information.

The ability to hold a bus transaction until the sP gives the approval to proceed is not only useful for Read-like transactions, but also for bus transactions like SYNC (memory barrier). The sP may need to carry out some actions before the SYNC bus transaction is allowed to complete in order to maintain the semantics of weak memory

[13] In more advanced bus protocols that supports out-of-order split address and data buses, such as the 6XX bus protocol, other mechanisms are potentially available. For instance, the address phase may be allowed to complete, with the data phase happening later; during this interval other bus transactions can begin and fully complete (*i.e.* both address and data phases finish) on the bus. This provision may, however, still be insufficient to prevent deadlocks. In particular this is unsafe if completing the address phase of a bus transaction means the processor cache takes logical ownership of cache-line, and will hold back other bus transactions attempting to Invalidate the cache-line until its data phase has completed. Determining whether deadlock can arise involves details that are still not uniform across bus protocol families. There is also close interaction between the snoopy bus and specifics of the directory base coherence protocol. Delving into this level of detail is beyond the scope of this thesis.

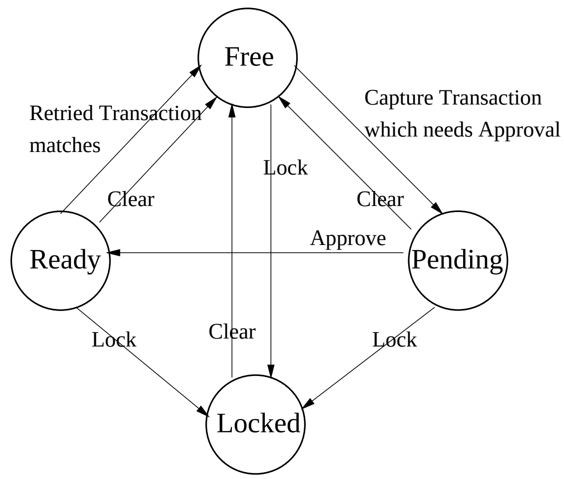


Figure 4-24: State transition diagram of the ApprovalReg.

models.

The NES Core provides an *Approval Register* (ApprovalReg) to coordinate this process. The ApprovalReg is best thought of as a transient cache entry – space is allocated when cache-miss is detected, the sP is responsible for filling the cache entry, and the cache-entry is freed after a single use. Although our implementation has only one ApprovalReg, more ApprovalReg's can be included to permit more outstanding pending approvals.

When a bus transaction requiring sP's approval first appears on the bus, and the ApprovalReg is not currently occupied, its details are recorded in the ApprovalReg and also captured into the Capture & Ack queue. The ApprovalReg state changes from "Free" to "Pending" as shown in Figure 4-24. Subsequently, the sP gives approval for the transaction to complete by writing the ApprovalReg, changing its state to "Ready". If data is involved, this write also supplies the SRAM address where the data should be read from or written to. When the same bus transaction is attempted again, it is allowed to complete. At the same time, the Approval Register is cleared.

Allowing the sP to specify an arbitrary data location SRAM address when it gives approval, as opposed to restricting the data to a fixed location, avoids copying if the data has arrived over the network and is buffered in some message receive

Notify sP?	Retry?	NES behavior
Yes	Yes	Approval Needed
Yes	No	Notification Only
No	Yes	Retry Only
No	No	Ignore (allow bus transaction to complete)

Table 4.2: The four possible NES responses to bus transactions to the sP Service Space or Snooped Space, and bus transactions with no associated address.

queue. Because the command to write the ApprovalReg comes from either the Local or Remote command queues, it is possible for a remote sP to return data directly to the NES Core without the involvement of the local sP. The local sP can be notified of such an event simply by setting the acknowledgement option in the command.

Service Space Response Table

The NES's response to bus transactions addressed to the sP Serviced space, or bus transactions without associated address (*e.g.* SYNC) is determined by a configurable response table, the Srv Space Response Table. The bus transaction type encoding is used as index into this table to obtain the response. As shown in Table 4.2 four responses are possible, arising from the cross product of two decisions: (i) whether the sP should be notified of the bus transaction, and (ii) whether the bus transaction should be retried. This level of programmability is only marginally useful for the sP Snooped space since only one or two responses make sense for most bus transaction types. It is included for uniformity of design because similar programmability is very useful for the Snooped Space support described in the next section.

Nevertheless, the design tries to make this generality as useful as possible. For instance, if the table indicates a response of Ignore to a read-like bus transaction, and there is no match against the ApprovalReg, data is read from a fixed SRAM location which system software can program with whatever *Miss value* desired. This feature allows aP software to determine whether a cache-miss has occurred – if the Miss value contains a bit pattern that is not valid data, software can check the value loaded to see if a cache-miss has occurred.

The response obtained from the Srv Space Response table is used only if a bus transaction does not match that captured in the ApprovalReg. Otherwise, the response is to retry the transaction if ApprovalReg is in the “Pending” state, and to allow it to complete if it is in the “Ready” state.

4.5.9 Snooped Space

The NES Core provides cache-line granularity access-permission check for a portion of the SMP's main memory, the Snooped (address) space. The mechanism enables the sP to selectively observe and intervene in bus transactions to this address space. This capability can be used to implement S-COMA style shared memory, or to allow fast access to the local portion of CC-NUMA shared memory[14]. It is also useful for sP implementation of fancier message passing interfaces, such as CNI, which can only be implemented efficiently if cache-line ownership acquisition is used to trigger processing.

The Snooped Space support shares many similarities with the sP Serviced Space mechanism but is more complex. NES response to a bus transaction to this address space is again configurable to any one of the four described in Table 4.2. Generation of this response is more complex, taking into account a 3-bit cache-line state associated with the bus transaction's address and maintained in a new memory bank, the clsSRAM (See Figure 4-26). Access to this memory is controlled by the Snooped Space unit in the aBIU. To simplify the design, the sP can write but not read this state with command issued via the Local or Remote command queues.

Other designs that maintain cache-line state bit for DRAM hardwire the interpretation of the state values [92]. In order to allow flexible experimentation with cache-coherence protocol, the StarT-Voyager NES uses a configurable table, the Snooped Space response table, to interpret the state values. Experimental flexibility is also the rationale for having 3 instead of 2 state bits per cache-line as having only four different cache-line states leaves little room for what the four states should mean.

[14] With small modifications to the aBIU FPGA, it can also be used to implement Nu-COMA style shared memory described in Section 2.2.

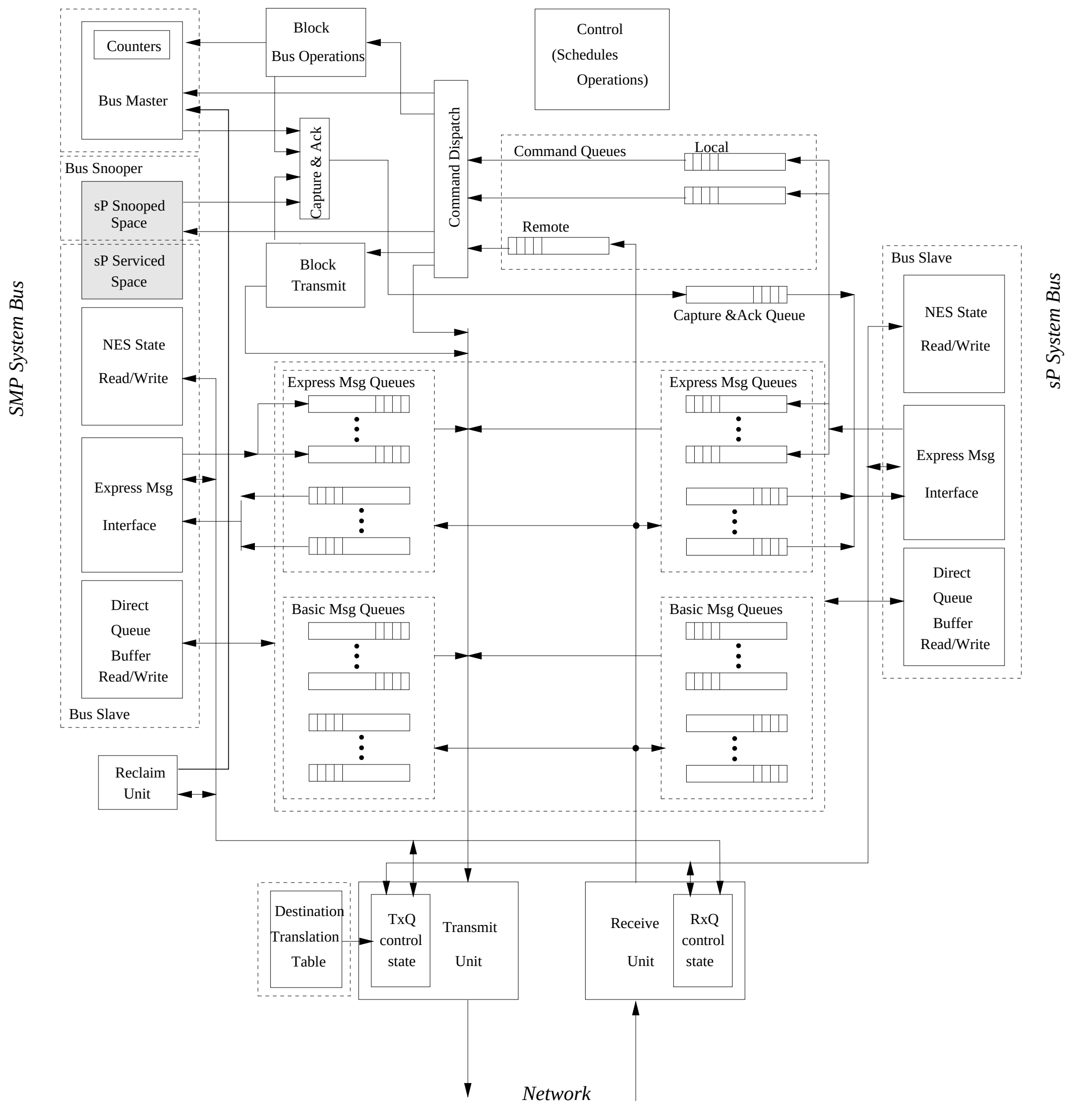


Figure 4-25: This diagram illustrates the addition of Snooped Space support through the shaded functional block. It also shows the full design of the StarT-Voyager NES.

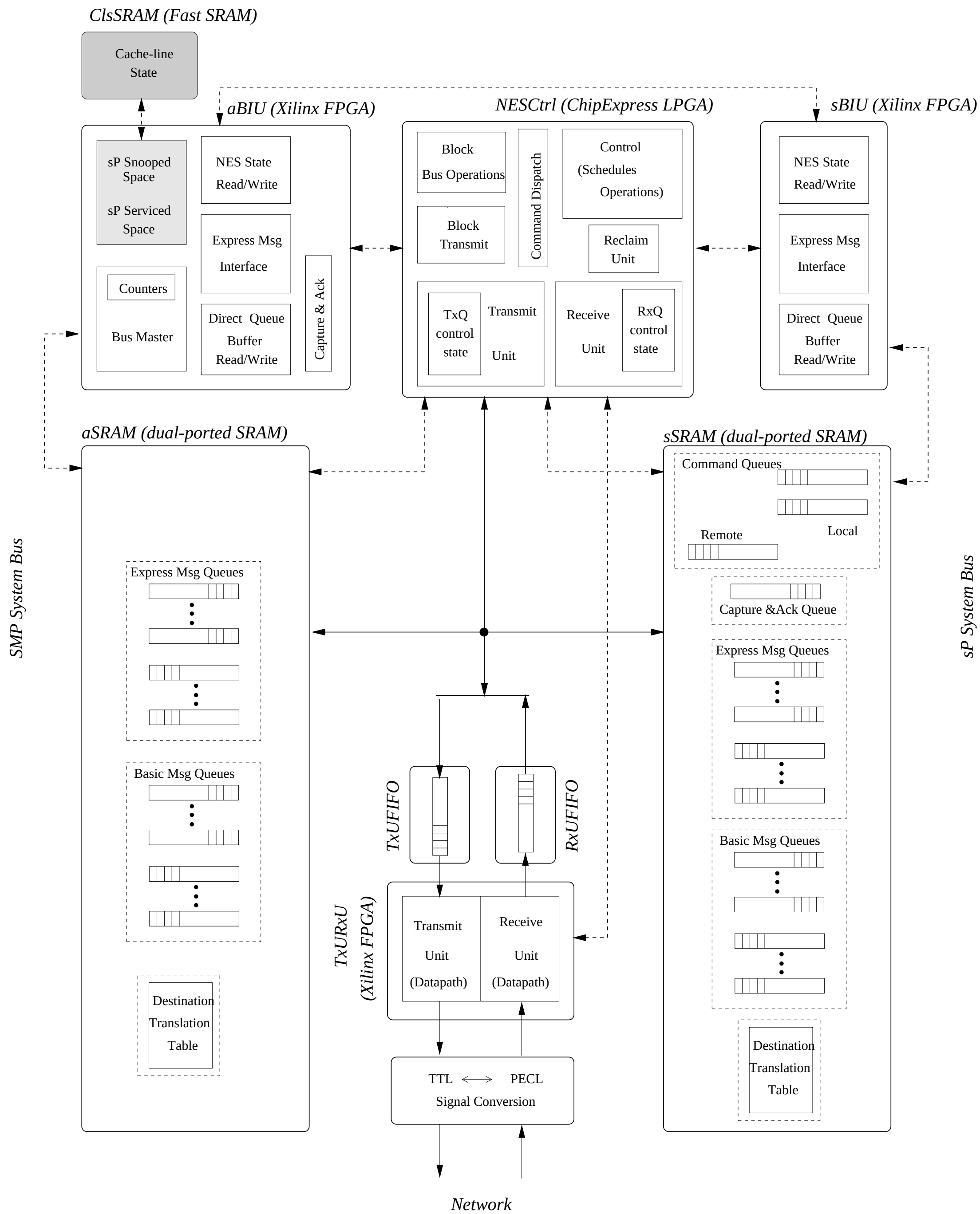


Figure 4-26: This diagram illustrates the addition of Snooped Space support through the shaded functional block from a device perspective. It also shows the full design of the StarT-Voyager NES.

The response to a bus transaction is determined by using its transaction type and its 3-bit cache-line state as offset into the Snooped Space response table. The result is a 2-bit response encoding the four possibilities described in Table 4.2.

The ApprovalReg described in the previous section is also used for Snooped Space bus transactions. As before, it filters out retries of a bus transaction requiring sP's approval, so that the sP is only informed once. Whenever the address of a bus transaction matches that in the ApprovalReg, the ApprovalReg's response supersedes that from the Snooped Space response table lookup. Again, a "Pending" ApprovalReg state retries the bus transaction while a "Ready" state allows the bus transaction to complete. In the latter case, the data supplied to a pending bus transaction is obtained from the NES SRAM location indicated by the ApprovalReg. Data for a write transaction is also directly written into that NES SRAM location.

Providing approval through the ApprovalReg is not the only way to end the retry of a Snooped Space bus transaction. An alternative is to modify the cache-line state kept in clsSRAM to a value which allows the bus transaction to complete. In that case, data for the bus transaction is read from or written to SMP main memory. If data is coming from or going to the network, using ApprovalReg has the advantage of avoiding cycling the data through main memory DRAM.

aP bus transactions initiated by the NES's bus master unit are treated specially by the NES Snooped Space unit. They are allowed to complete regardless of the status of the ApprovalReg, the various response tables, and the relevant clsSRAM cache-line state value.

The use of configurable response table adds to the latency of response generation. But because the table is small, this increase does not present a serious timing issue in our design. A production system that supports only one or a small number of fixed protocols will not need this flexibility.

The addition of Snooped Space support brings us to the full design of the StarT-Voyager NES. Through incrementally adding more functions to the design, this section both provides a detailed view of the NES micro-architecture, and illustrates the re-use of existing functional blocks as new capabilities are added to the NES.

4.6 Mapping onto Micro-architecture

Now that the NES micro-architecture has been described, this section shows how the macro-architecture described in Chapter 3 is realized on the micro-architecture.

4.6.1 Physical Network Layer Implementation

The Arctic network closely matches the requirements of the Physical Network layer. The only missing property is an active means of bounding the number of outstanding packets. To keep the design simple, we rely on the maximum buffering capacity of the entire Arctic network to provide this bound instead of introducing some new active mechanism in the NES. Our system is relatively small with a target size of 32 nodes. This translates into (5×32) Arctic routers. Buffering capacity in each Arctic router is also not large, (12×96) Bytes. This gives a total network buffering capacity of 180 kilobytes. This bound is low enough to be a useful overflow buffer size estimate for Reactive Flow-control.

4.6.2 Virtual Queues Layer Implementation

The StarT-Voyager NES implements the Virtual Queues Layer and the Application Interface Layer with a combination of NES Core hardware and sP firmware. The division of tasks between hardware and firmware is mostly based on function, with those expected to be used infrequently delegated to sP firmware. Some functions, such as those related to Basic, Express message passing mechanisms and their TagOn variants, are performed by both NES Core hardware and sP.

Resident and Non-resident Queues

The NES Core hardware provides fast implementation of a limited number of message queues, a number which is meant to capture the typical working set. The sP implements, at lower performance, a much larger number of queues to meet the design goal of supporting a large number of simultaneously active queues. The former, referred to as Resident message queues act as a sP firmware managed cache of the latter, the Non-resident message queues. Switching a logical message queue between Resident and Non-resident resources is a local decision requiring no coordination with other nodes and is transparent to aP software.

In the case of Resident message queues, the NES Core hardware directly implements the Virtual Queues Layer functions of destination translation, and multiplexing and demultiplexing messages from several hardware message queues onto the Physical Network Layer services. This is fairly straight forward, and is described in the micro-architecture Sections 4.5.1 through 4.5.3.

To logically support a larger number of message queues, the sP firmware multiplexes the Non-resident queues onto a subset of the Resident queues. (Discussion of exactly how the aP interacts with the sP to use Non-resident queues is deferred until next Section when the Application Interface Layer mapping is described.) During transmit, the sP performs destination translation by either physical addressing of packet destination, or changing translation table entry. Low-level design choices, such as associating a source identity with each translation table entry instead of each message queue, makes it feasible to use the latter approach. Other NES Core functions, such as giving sP the ability to initiate aP bus transactions to move data between aSRAM and sP DRAM, and reading data (message header) from aSRAM, are also necessary capabilities.

The NES Core hardware's RQID cache-tag lookup mechanism (Section 4.5.1) makes it possible to switch receive queue between Resident and Non-resident queues without global coordination. It also channels packets of Non-resident queues to the Miss/Overflow queue where sP takes over the task of sorting the packets into their final receive queues.

Reactive Flow-control

Reactive flow-control, for preventing deadlock, is implemented in sP firmware. For Non-resident queues, it is clear that the sP, which is already handling all the messages, can impose Reactive flow-control, both triggering throttle, getting out of it, and selectively disabling transmission to a particular destination queue.

For Resident queues, the sP is also in a position to impose Reactive flow-control as long as the threshold water-mark, the point at which throttle is triggered, is larger than the receive queue size. When a Resident receive queue overflows, overflowing packets are diverted to the Miss/Overflow queue serviced by the sP. The sP therefore starts handling the Resident queue's in-coming packets as well, and can initiate throttle when necessary.

The NES SRAM buffer space of Resident receive queue is actually only part of the total buffer space for the receive queue. The overflowing packets are buffered in memory outside NES SRAM, in aP DRAM or sP DRAM. This is also the buffer space for the queue when it operates in Non-resident mode. Ideally, the threshold water-mark should match the size of the Resident receive queue's buffer space in NES SRAM, so that as long as a program operates within this receive queue buffering requirement, the sP is never involved and end-to-end message passing latency is not degraded.

When a Resident receive queue overflows, the sP programs the NES Core to notify it of subsequent bus transactions accessing this queue's state[15]. In this way, the sP can move packets from the overflow queue back into the aSRAM as space frees up, and monitor whether space usage in the Resident receive queue has dropped below the low water-mark.

The sP is also involved in implementing selective disable of Resident transmit queues, a function not directly supported in hardware. To selectively disable a destination, the sP modifies the physical destination of its translation table entry so that packets heading to this logical destination are directed back to the sP itself. When the sP receives such looped-back packets, it buffers them up for subsequent re-transmission. In this way, the Resident transmit queue, which operates in a strict FIFO manner, can continue to send other packets out to other destinations. In the event that too many packets have looped back, the sP can shut down the transmit queue, by writing its queue-enable bit, to avoid running out of loop-back packet buffer

[15]This was not implemented in the NES, but it requires only simple modifications to the aBIU FPGA Verilog code.

space.

4.6.3 Application Interface Layer Implementation: Message Passing Interfaces

As pointed out in the last section, the message passing interfaces, with the exception of DMA, are implemented with both Resident and Non-resident queues. The Application Interface Layer portion of Resident queues is completely implemented in NES Core hardware, as described in Sections 4.5.1, 4.5.2, and 4.5.4. DMA implementation is also described earlier in Section 4.5.7. This section describes how the interfaces of Non-resident queues are implemented.

A key component of Non-resident message queue implementation is the mapping of addresses. This determines the visibility of access events to the sP, the level of control sP exercises over such events, and the message data path efficiency.

Non-resident Basic Message Queues

The message queue buffers of Non-resident Basic message queues are mapped to aP main memory, addresses for reading pointer values are mapped to NES SRAM[16] and addresses used for updating pointers are mapped to the sP Serviced Space. Mapping pointer update addresses to sP Serviced Space has the advantage of relieving the sP from continually polling memory locations holding queue pointers. Instead, captured transactions trigger sP processing. This is crucial as the sP may otherwise have to blindly poll many locations.

Updates to the producer pointer of a Non-resident Basic message transmit queue causes the sP to attempt a message transmit. If the proxy Resident transmit queue, *i.e.* the queue onto which Non-resident messages are multiplexed, has sufficient buffer space, the sP issues a bus operation command to read message data from the SMP system bus into the proxy queue's next available buffer space. The sP will require

[16]Because protection is only enforceable at 4 kByte granularity, but only 8 bytes is really needed for message queue state, the NES SRAM can be designed with a window where each 4 kilobyte page is aliased to only 8 bytes of NES SRAM space.

notification of command completion, so that it can then read in the message header, translate it, and update the proxy queue's producer pointer to launch the packet into the network. At that time, it also updates the emulated queue's consumer pointer to free up transmit buffer space. If the sP is unable to transmit the message immediately because the proxy transmit queue is full, the pending send is recorded and subsequently resumed when transmit space frees up. The sP periodically polls the proxy transmit queue's state to determine if sufficient space has freed up.

Non-resident Express Message Queues

The address used to indicate an Express message transmit is mapped to sP Serviced space. This is needed for implementing FIFO semantics, *i.e.* a store has further effects beyond merely over-writing the data in a fixed memory location, and allows event driven sP processing. Similarly, the address for receive polling is also mapped to sP Serviced space. Buffer space for the non-tagged-on portion of Express message is provided in sP DRAM, while buffer space for the tagged-on data is mapped to aP DRAM.

Mapping the Express receive polling address to sP Serviced space provides the required functionality but the aP incurs fairly long latency when polling for a message. A design which takes the sP off the latency path of providing the polled data is highly desired. This is tricky, however, because although the sP can deposit the next message at the address that aP software polls from, a mechanism is needed to ensure that the aP software sees this message only once. The generality of OnePoll also complicates this as several different polling addresses could all receive from the same receive queue.

If OnePoll is limited to only the high and low priority receive queues of a typical Express message End-point, and a simple enhancement is made to NES Core hardware, a lower latency alternative is possible. The required enhancement enables the Snooped Space support to not only read clsSRAM state-bit, but also modify it. A new configurable table could be introduced to determine the next clsSRAM value based on the current clsSRAM value. With the Express message receive address mapped to a Snooped Space location, the sP can get the next answer ready there, so that an aP can poll, *i.e.* read, it from DRAM. Because the aP needs to read this cache-line exactly twice (assuming aP uses only 32 bit reads to receive Express messages), the automatic clsSRAM value transition will need to go from one which generates an Ignore response (written by sP), to a second one that Notifies sP (automatic), to a third which Retries without Notifying sP (automatic).

This enchancement scheme works even when OnePoll-ing is from both receive queues of an end-point because the address for polling only one receive queue and that for polling both receive queues in an end-point all fall within the same cache-line. Unfortunately, the more general form of OnePoll does not work in this enchanced scheme. Although this new scheme improves the latency of a single Express receive poll, the minimum interval between two polls, and hence the maximum throughput, is still constrained by sP processing.

4.6.4 Application Interface Layer Implementation: Shared Memory Interfaces

Coherent shared memory implementation on StarT-Voyager relies heavily on the sP's involvement. For instance, as described below, the sP is involved in servicing all cache-misses[17] under both CC-NUMA and S-COMA style shared memory. The sP implements most of the functions of the cache protocol engine and the home protocol engine.

The basic NES Core hardware mechanisms that enable the sP to implement cache-coherent distributed shared memory are described in Sections 4.5.8 and 4.5.9. It is beyond the scope of this thesis to design and implement complete coherence protocols. In this section, we sketch out a cache-miss servicing example for S-COMA style shared memory to illustrate how coherence protocols can be put together.

In this example, an S-COMA style cache-line miss results in fetching data from the home node. The NES cache-line state and the response table entries are set up

[17] We use the term "cache-miss" broadly to mean not having the correct access permission, *i.e.* acquiring write permission is considered a cache-miss.

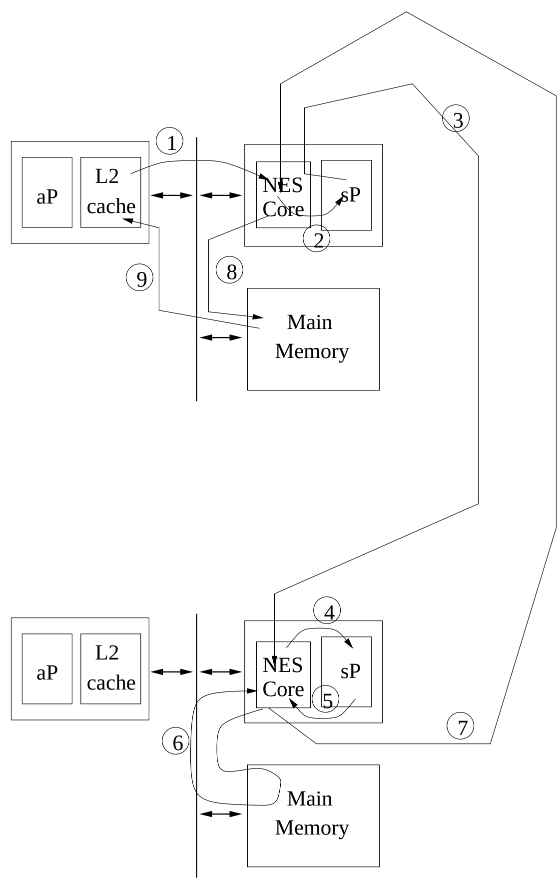


Figure 4-27: S-COMA style processing of a READ cache-miss on StarT-Voyager. In this example, clean data is available in the home node.

so that a read transaction to an absent cache-line is retried while the sP is notified. As described in Section 4.5.9, when such a bus transaction is attempted, the NES Core hardware notifies the sP of the event by sending it details of the bus transaction through the Capture & Ack queue. It also records details of the bus transaction in the ApprovalReg (Step 1 in Figure 4-27).

The top-level sP code is an infinite loop, in which the sP polls for events from the Capture & Ack queue, the overflow queue, and other message receive queues used for inter-sP communication. OnePoll is used to improve polling efficiency. The sP may also need to monitor for space in transmit queues if there are pending message transmits.

When the sP reads the captured read transaction information (Step 2 in Figure 4-27) and decodes it as an S-COMA space read cache-miss, it translates the captured physical address into a global address and a home node number. Using the Express message mechanism, the sP sends a request message to the home node (Step 3). This message could be sent through a Local Command queue or another Express message queue. The choice will be the former if the protocol relies on ordering between this new request and earlier messages sent through the Local Command queue. The sP will typically also need to keep some transient information about the pending bus transaction in order to deal with in-coming commands which may be directed at the same cache-line address.

Action now shifts to the home node. The home node sP receives this message when it polls for new events in its top-level polling loop (Step 4a). This sP looks up the cache-line's directory information to decide what to do. In our example, the directory state indicates that the cache-line is clean at home; consequently data can be supplied directly from the home node. To achieve this, the home sP first translates the global address into a local physical address, and then issues a bus master command (Step 5) to transfer data from that local SMP system bus physical address into NES SRAM (Step 6). Through the same local command queue, the sP also issues an Express-TagOn message to send that data to the requesting sP. Ordering imposed by NES Core hardware between bus master command and Express/Express-TagOn message command ensures that the second command is not executed until data fetched by the first is in NES SRAM.

While waiting for this reply, the sP at the requesting node modifies the clsSRAM state of the cache-line with the pending transaction to one which allows read (Step 4b). Because the ApprovalReg still records the pending transaction and its response dominates, the READ bus transaction is still unable to proceed. Once the reply comes in (Step 7), the sP moves the data from the receive queue buffer into the appropriate aP DRAM location (Step 8) using a bus master command issued through a Local Command queue. It then clears the ApprovalReg, and a retried READ bus transaction can now complete (Step 9).

Discussions

The above scenario involves the sP three times: twice at the requesting node, and once at the home node. The last involvement of the sP can be removed, or at least taken off the (latency) critical path of returning data to the pending bus transaction. To achieve this, the home sP sends the reply into the Remote Command queue, using a reply message similar to a DMA packet. The requesting sP can be informed of the reply using the acknowledgement option on this bus command.

Heavy sP involvement in cache-miss processing has both advantages and disadvantages. The main disadvantage is relatively long cache-miss latencies; therefore, shared memory performance on this machine is very sensitive to cache-miss rate. The advantage is flexibility in how the protocols are implemented, including the option of trying out less common ways of maintaining coherence which may improve cache-miss rate. Given the tradeoffs, this design makes sense for an experimentation platform. An actual work-horse design should include more custom hardware support to reduce sP involvement during cache-miss processing.

An interesting question is what other NES Core mechanism can be provided to reduce sP involvement in cache-miss processing without drastically affecting the flexibility of using different coherence protocol. One avenue is to remove the duty of cache protocol engine from the sP, or at least only involve the sP in difficult but rare

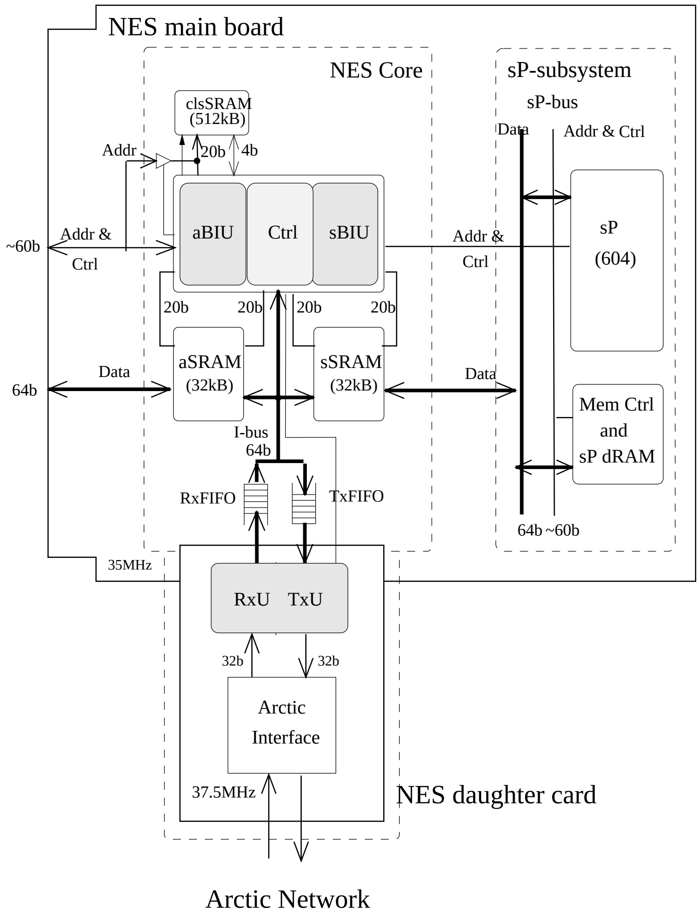


Figure 4-28: Major components of the StarT-Voyager NES.

cases when ordering of events is ambiguous. We need to experiment with a number of protocols using the existing StarT-Voyager NES design before we can answer that question concretely.

4.7 NES Hardware Implementation

The StarT-Voyager NES is physically separated into two printed circuit boards: the NES main board, and the NES daughter card as shown in Figure 4-28. Logic specific to the network are isolated in the daughter card so that porting the design to another network involves replacing only the daughter card.

The main board contains two major portions: the sP subsystem and the NES Core. The former consists of a PowerPC 604 microprocessor [46], employed as an

Component	Implementation Technology	Comments
NESCtrl	ChipExpress CX2001 Laser Programmable Gate Array	approximately 55k gates plus 1.5 kilobits of ram.
aBIU	Xilinx XC4052XL, -09 speed grade	47% of CLB's used.
sBIU	Xilinx XC4036XL, -1 speed grade	29% of CLB's used.
TxURxU	Xilinx XC4036XL, -1 speed grade	31% of CLB's used.

Table 4.3: Custom designed hardware in the StarT-Voyager NES.

embedded processor, and its own memory system comprising of the commercially available MPC 105 memory controller[76], and ordinary DRAM.

The NES Core includes a control block (NesCTRL) implemented with a ChipExpress LPGA [20], two Bus Interface Units (aBIU and sBIU) implemented in Xilinx FPGA's and two banks of dual ported SRAM. One bus interface unit, the aBIU, interfaces to the SMP's memory bus, while the other, the sBIU, interfaces to the sP subsystem's memory bus.

Porting this design to a different SMP will require changes to the aBIU and possibly sP firmware, but other parts of the NES hardware can remain unmodified. Originally, a single LPGA is to contain the NESCtrl, aBIU and sBIU. This introduces pin-count constraints so that only address and control signals from the two external buses are directly wired to the BIU's. The 64-bit wide data portion of the buses connects directly to the dual ported SRAM banks only. We subsequently moved the aBIU and sBIU into large FPGA's which no longer has the pin-count constraints, but we left this design unchanged.

The main data-path through the NES main board is eight bytes[18] wide. Each of the dual ported SRAM banks, aSRAM and sSRAM, has 32 kilobyte of storage provided by eight, byte-sliced, IDT70914s dual-ported, synchronous, SRAM memory chips. Five IDT709269s hardware FIFO chip, each 36 bit wide and 512 deep, are used

[18] There are eight more parity bits, but we will ignore parity in this discussion.

to implement the RxFIFOF and TxFIFO.

4.7.1 Design Flow

The main custom designed hardware components in the StarT-Voyager NES are the NESCtrl, aBIU, sBIU, and TxURxU. Design of these components were done in Verilog at the Register Transfer Level (RTL) and then automatically synthesized, placed and routed with CAD tools.

The NESCtrl LPGA, designed by Daniel Rosenband, is synthesized with Synopsys's Design Compiler, and then placed and routed by ChipExpress with proprietary tools. The TxURxU FPGA, designed by Michael Ehrlich, is synthesized with Synopsys's FPGA Compiler and then placed and routed with Xilinx's M1 tool set. The aBIU and sBIU FPGA's, designed by me, are synthesized using Synplicity's Synplify, and again placed and routed with Xilinx's M1 tool set.

Functional simulation and testing is done using the Cadence Verilog XL environment, augmented with our own C code to model the aP and sP processors and DRAM. The C code interacts with Verilog through Cadence's PLI (Programming Language Interface). Verilog code implements the bus interface portion of the processors, while C code models the rest of the processors. As for memory, the Verilog code implements the memory controls, but the actual data storage elements are modeled in C code. Functional testing is done with hand coded test examples; these include tests targeting specific functions, as well as random combinations of tests.

Chapter 5

Evaluations

This chapter describes simulation evaluation results obtained with micro-benchmarks. The results fall into three categories. The first category evaluates StarT-Voyager NES's support for multiple message passing mechanisms. The goal is to demonstrate that additional mechanisms layered on top of a basic set of capabilities improve performance by taking advantage of specific communication characteristics, such as very small or very large message sizes. The second category examines the multiple message queues support, quantifying the cost of supporting a large number of message queues in the StarT-Voyager design. The third category evaluates the performance of the off-the-shelf sP, providing both absolute performance numbers for representative communication operations, and an understanding of the factors limiting its performance.

Although actual hardware was designed and built as part of this research work, it was not available in time for the evaluation reported here. Instead, the experiments presented are conducted on a simulator, StarT-sim, described in the next section. Although we were unable to validate the simulator against actual working system due to late availability of working hardware, we expect the conclusions drawn from our results to hold. This is because we base our conclusions on *relative* performance relationships, which still hold even if there is inaccuracies in the simulator since those affect the performance numbers uniformly.

5.1 Evaluation Methodology

StarT-sim is an execution driven simulator; it "executes" the simulated application during each simulation run so that the program can respond to "run time" conditions. Thus, it is possible for simulation of the same program to follow different execution paths when run time conditions such as memory access or message passing latency vary. Written in C, StarT-sim runs in a single process and has two major components: (i) a processor core simulator (AugRS6k), and (ii) a memory system, NES and network simulator (Csim).

5.1.1 Processor Core Simulation

The processor core simulator, AugRS6k, operates in an event driven fashion and is capable of modelling a multi-processor system. In the interest of faster simulation speed, the processors are not simulated in lock-step cycle by cycle. Instead, as long as a processor is executing instructions that do not depend on external events, it advances its "local" time and continues executing more instructions. In our system, this means that simulation of a processor only suspends at a memory access instruction. At that point, an event to resume simulating the processor when global time reaches its local time is enqueued in the global event queue.

Processor simulation is achieved by editing the assembly code of the simulated program with an extra custom compilation pass. Memory access instructions are replaced with calls to special routines which enforce causal order by suspending and resuming simulation as appropriate. These routines also make calls to model caches, the NES, and the network. Code is also added at basic block exit points to advance local processor time.

StarT-sim models the latencies of memory access instructions precisely. These are determined dynamically, taking into account cache-hit/miss information, and the latency and contention effects in the memory system and the NES. StarT-sim's modelling of processor core timing is approximate. It assumes that each non-memory access instruction takes one processor cycle. There is no modelling of superscalar instruction issue, data-dependence induced stalls, or branch prediction and the cost of mis-prediction. Modelling the processor more accurately is highly desired but it requires an effort that is beyond the scope of this thesis.

The processor simulator was based on the Augmint[79] simulator which models x86 processors. Initial effort to port it to model the PowerPC architecture was done at IBM Research. We made extensive modifications to complete the port, and to model the StarT-Voyager system accurately.

5.1.2 Memory System, NES and Network Simulation

The second component of StarT-sim, Csim, is a detailed RTL (Register Transfer Level) style model of the processor cache, memory bus, NES, and Arctic network written in C[1]. Simulation in Csim proceeds in a cycle-by-cycle, time synchronous fashion. Modelling is very detailed, accounting for contentions and delays in the NES and the memory system. The NES portion of this model is derived from the Verilog RTL description used to synthesize the actual NES custom hardware.

StarT-sim models two separate clock domains: a processor clock domain and a bus clock domain. The former can be any integral multiple of the latter, and is set to four times in our experiments to reflect the processor core to bus block ratio of our hardware proto-type. This ratio is also common in systems sold today (end 1998); for example, a typical system with a 400 MHz Pentium II processor employs a 100 MHz system bus. The processor core, load/store unit, and cache are in the processor clock domain while the memory bus, the NES, and the network operate in the bus clock domain. The performance numbers reported in this chapter assume a processor clock frequency of 140 MHz, and a bus clock frequency of 35 MHz.

StarT-sim also models address translations, including both page and block address translation mechanisms found in the PowerPC architecture. Our simulation environment provides a skeletal set of virtual memory management system software routines to allocate virtual and physical address ranges, track virtual to physical

[1]Csim is mainly the work of Derek Chiou. I wrote code to model some of the NES functions and assisted in debugging Csim.

address mappings, and handle page faults.

5.2 Multiple Message Passing Mechanisms

This section presents a detailed micro-benchmark based performance evaluation of Resident versions of Basic message, Express message, Express-TagOn message, and DMA. Performance is measured in terms of bandwidth, latency and processor overhead. The result shows that while Basic message presents a versatile message passing mechanism, adding thin interface veneers on it to support the other message passing mechanisms significantly enhances performance and functionality for certain types of communication. Together, the collection of mechanisms is able to offer better performance over a range of communication sizes and patterns than is possible with Basic message alone. A synopsis of specific instances follows.

NES hardware supported DMA presents significant advantage for block transfers beyond about 1 kByte. If such transfers are done using Basic message, the overhead that aP code incurs while copying message data is the dominating performance constraint. Moving message through the aP registers is an unnecessarily convoluted data path in this case. This inefficiency imposes performance limits that are significantly lower than those under hardware DMA – bi-directional block transfer bandwidth under Basic message is only about a third of that under hardware DMA. Hardware DMA also brings with it the equally significant advantage of off-loading the data packetization overhead from the aP.

When message latency is critical and message payload is very small, as often happens with urgent control messages, Express message is the best option. Intra-node handshake overhead between aP software and the NES is a significant component of latency. In StarT-Voyager, this amounts to about 45% of the overall latency for the smallest Basic message. This ratio is expected to be even higher for future system because the latency through the StarT-Voyager NES and Arctic network are not the best possible. The NES is a board level design which incurs latencies that an integrated ASIC can avoid. Newer networks, like SGI's Spider [33], have significantly

Message Passing Mechanism	Bandwidth (MBytes/s)
Express Message	9.51
Basic Message	53.15
Express-TagOn Message	55.31
NES Hardware DMA	84.40

Table 5.1: Bandwidth achieved with different message passing mechanisms to transfer 4 kByte of data from one node to another.

lower latency than Arctic. With Express message, the cost of intra-node handshake is reduced drastically, so that overall latency is close to 30% lower than that for Basic message.

When multi-casting a message, TagOn message offers the best mechanism by avoiding redundantly moving the same data to the NES multiple times. Through more efficient intra-node data movement and control exchange between the aP and the NES, TagOn message both reduces the aP processor overhead and improves throughput. TagOn's interface is also "thread safe", and is thus a better option if a message queue is shared between multiple threads.

The remainder of this section, Sections 5.2.1 through 5.2.3 presents details of the benchmarks, performance numbers and explanation of these numbers.

5.2.1 Bandwidth

Table 5.1 shows the bandwidth achieved using the four message passing mechanisms. As expected, NES hardware implemented DMA achieves the highest bandwidth, while Express message delivers the lowest bandwidth. The bandwidth for Basic and Express-TagOn are effectively the same.

The bandwidth advantage of NES hardware DMA is even more pronounced when bi-directional transfer bandwidth is considered, *i.e.* two nodes are simultaneously exchanging data with each other. Hardware DMA sustains a combined bandwidth of 166.70 MBytes/s for 4 kByte bi-directional block transfers. The corresponding bandwidth for the other three transfer mechanisms is actually lower than their respectively uni-directional transfer bandwidth since the aP has to multiplex between message send and receive actions.

Micro-benchmark Details

Unidirectional bandwidth is measured with different send and receive nodes. The time taken to transfer 4 kBytes of data from the sending node's main memory to the destination node's main memory is measured and bandwidth is derived accordingly. All benchmarks are written in C.

For NES hardware supported DMA, the initiating aP sends its local sP an Express-TagOn message with details of the node to node transfer request. This sP coordinates with the remote sP to translate virtual addresses into physical addresses, and set up the NES DMA hardware. From that point on, all the data transfer tasks are handled by NES hardware, which also notifies the destination sP when the transfer completes.

In the Basic Message block transfer micro-benchmark, the source aP is responsible for packetizing and marshalling data into transmit buffers. It also appends a destination address to each packet. Because each packet is "self identifying", the destination aP maintains almost no state for each in-progress DMA. The destination aP is responsible for copying data from receive queue buffers into the target main memory locations. A similar scheme is used by the Express-TagOn block transfer micro-benchmark.

Carrying the destination address in each packet simplifies the destination aP's job but consumes 4.5% of the total payload. An alternate design avoids this overhead by having the destination node generate destination addresses. This, however, requires establishing a connection between the source and destination node. Furthermore, to avoid the overhead of sequence numbers and connection identification in each packet, the data packets have to arrive in the sent order, and only one connection can exist between each source-destination pair at any time.

This connection based approach is used in the Express message block transfer micro-benchmark. The extremely small payload in Express message makes it impractical for each packet to be self identifying. By adopting the connection approach, the

Block Transfer Size	Bandwidth (MBytes/s)
2 kBytes	76.8
4 kBytes	84.4
8 kBytes	88.6

Table 5.2: Measured NES hardware DMA bandwidth as transfer size changes. Bandwidth improves with larger transfers because of fixed setup overhead.

benchmark can transfer 4 byte in each data packet. These packets all belong to the same order set to ensure the desired arrival order. Each Express message has room to carry 5 more bits of information, which is used to identify the connection number and the packet type – connection setup vs data packets.

Bandwidth Limiting Factors

The NES hardware DMA bandwidth is limited by a combination of Arctic network bandwidth limit and the NES DMA architecture. Each NES hardware DMA data packet carries 64 bytes of data, and 16 bytes of aP bus operation command. There is a further 8 bytes of overhead imposed by the Arctic network. Thus, data takes up only 64/88 of each packet. Consequently, although Arctic's link bandwidth is 140 MBytes/s, our DMA scheme is bounded by a maximum bandwidth of 102 MBytes/s. The overhead of setting up the NES DMA hardware is responsible for bringing that number down to the measured 84.4 MBytes/s for 4 kByte transfers. Because this overhead is constant regardless of the transfer size, the measured bandwidth improves with larger transfer sizes, as illustrated in Table 5.2.

We can model the time taken for an n byte transfer as: $n \times x + c$. Using the timing for 2, 4 and 8 kbyte transfers, we arrive at these values for x and c:

$$x = 10.7 \times 10^{-12}$$

$$c = 4.6 \times 10^{-6}$$

As $n \to \infty$, the achieved bandwidth is:

$$
\begin{aligned}
\lim_{n\to\infty}(\frac{n}{nx+c}) &= \lim_{n\to\infty}(\frac{1}{x} - \frac{c/x_2}{n+c/x}) \\
&= \frac{1}{x} \\
&= 93.5 \text{ MBytes/s}
\end{aligned}
$$

c is the fixed coordination overhead that works out to be 4.6 μs.

The other three message types all share the same bandwidth limiting factor: the aP's processing overhead. The handshake between the aP and the NES, and marshalling data through the aP to the NES limits the bandwidth attained. Much of this is due to aP bus access cost. The next paragraph justifies this claim in the case of the Basic Message.

Table 3.1 in Chapter 3 estimates the number of bus transactions for each Basic Message packet. Under steady state in the block transfer micro-benchmark, the handshake aggregation feature of Basic message is not expected to help reduce cost, neither in obtaining free buffer, nor in updating transmit queue producer pointer. This is supported by the fact that Express-TagOn achieves the same (in fact slightly better) bandwidth as Basic Message – the two mechanisms have almost identical overheads once aggregation is not done in Basic Message.

A maximum size Basic message – 3 cache-lines – is therefore expected to incur about 9 bus transactions, which occupies the bus for 42 bus clocks. This is close to the one packet every 53 bus clocks throughput inferred from the bandwidth of the Basic Message block transfer benchmark. The difference between this number, and the memory bus occupancy number is due to other aP processor overhead – buffer allocation and deallocation and data marshalling.

Message Passing Mechanism	Minimum Size Message Latency	
	# pclk	μs @ 140MHz pclk
Express Message	234	1.67
Express-TagOn Message	414	2.96
Basic Message	328	2.34

Table 5.3: One-way message latency for minimum size Express, Express-TagOn, and Basic messages. Numbers are reported in both processor clocks (pclks), and in microseconds (μs) assuming a processor clock of 140 MHz. The latency is measured from the point when sender aP software begins execution to the moment the receiver aP software reads in the message data. It includes the software cost of allocating sender buffers. The reported numbers are for communication between nearest neighbors; each additional hop on the Arctic network adds 18 pclks under no-contention situations.

5.2.2 Latency

Latency is measured by ping-ponging a message between two nodes a large number of times, averaging the round trip time observed by one node, and then halving that to obtain the one-way message latency. We are only interested in the latency of short messages that fit within one packet since the latency of multi-packet messages is reflected in the bandwidth benchmarks described previously. For this reason, we present data for Express, Express-TagOn and Basic messages, but not for hardware DMA.

Table 5.3 lists the one-way latency of sending a minimum size message under different message passing mechanisms[2]. The result shows that Express Message achieves latency 30% lower than that of Basic Message, while Express-TagOn incurs the longest latency due to its limited message size options.

As the message size increases from two to twenty words (4-byte words), the latency of Basic messages increases in a step fashion. Each step coincides with the message buffer crossing a cache-line boundary: between 6 and 8 words, and then again between

[2]This may appear to be an unfair comparison since the minimum message size of Express-TagOn is larger than those of Express and Basic messages, owing to its paucity of message size option. The intent of this table is, however, to capture the minimum latency incurred when the payload is extremely small.

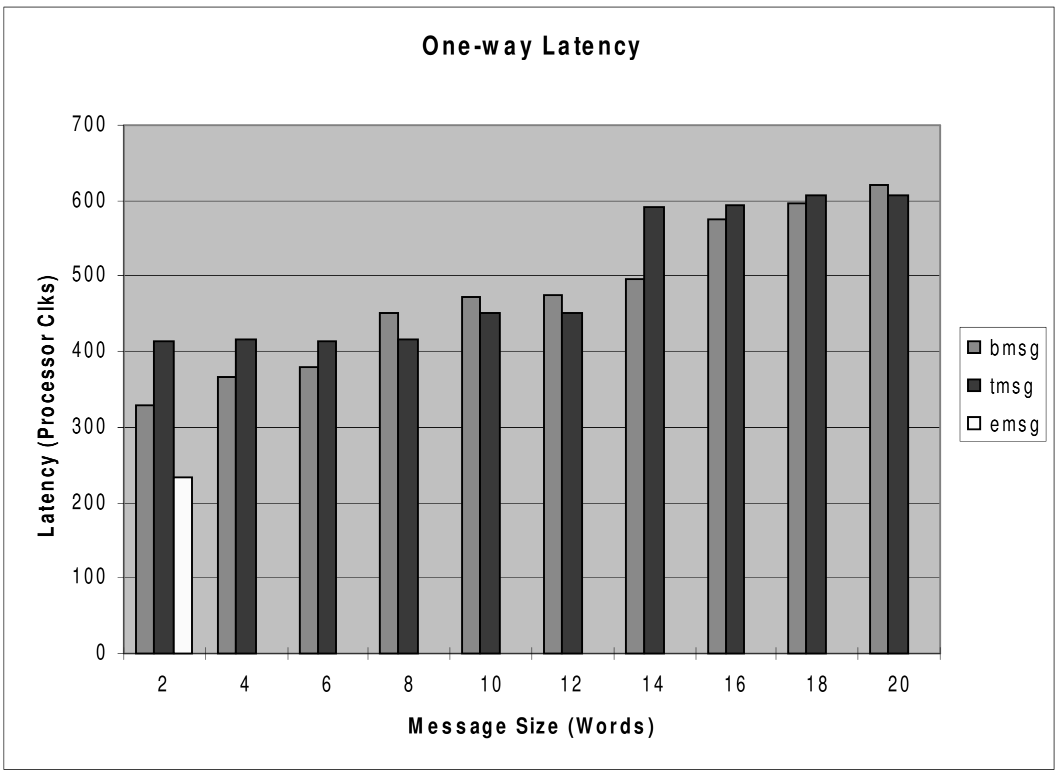


Figure 5-1: One-way latency for Express (emsg), Express-TagOn (tmsg) and Basic (bmsg) messages as message size varies from 2 to 20 (4 byte) words.

Latency components	Express (proc clks)	Express-TagOn (proc clks)	Basic (proc clks)
aP software send	22	100	70
Delay through source NES	80	80	72
Delay through Arctic network (1 switch)	56	56	56
Delay through destination NES	48	48	48
aP software receive	28	130	82
Total	234	414	328

Table 5.4: Breakdown of end-to-end latency of minimum size Express, Express-TagOn and Basic messages. The aP software components include latency on the memory bus.

14 and 16 words. This is illustrated in Figure 5-1 which also shows Express-TagOn message latency. Express-TagOn message latencies also have a grouping pattern determined by cache-line boundary. But owing to its different interface, the cache-line boundary for Express-TagOn message occurs between message sizes of 12 and 14 words.

Table 5.4 provides the breakdown of the end-to-end latency for minimum size messages. aP software overhead, including moving the data to and from the NES, dominates the latency of Express-TagOn and Basic messages. Of the two, Express-TagOn incurs higher software overhead since its buffer management is more complex – it has to deal with the buffer space used for the TagOn data, in addition to that for its Express-style header portion. Together, these numbers clearly show that intra-node overhead is a serious issue for message passing. With Express message, we are able to pare this down to the point where the latency through StarT-Voyager's hardware components dominate.

The absolute latency through the NES and Arctic is quite substantial for all three message types. It accounts for the greater part of Express message latency – 184 out of 234 processor clocks. There are three reasons for this, all of which can be overcome with better hardware implementation technology.

Firstly, the 4:1 processor core to bus clock ratio greatly magnifies any NES or network latency. This can be improved by using better silicon technology in the NES

Message Passing Mechanism	Tx Processor Overhead		Rx Processor Overhead	
	# pclk	μs @ 140MHz pclk	# pclk	μs @ 140MHz pclk
Express Message	9	0.06	28	0.20
Express-TagOn Message	73	0.52	116	0.83
Basic Message	49	0.35	49	0.35

Table 5.5: Processor overhead for minimum size message (one 4-byte word).

and the network to bring their clock speeds closer to that of the processor. Secondly, our NES design is a loosely integrated board-level design. Loose integration results in our NES message data path crossing 4 devices between the Arctic network and the aP system bus. Each chip crossing adds at least two cycles to latch-in and latch-out the data. Implementing the entire NES Core in a single ASIC will improve this. Lastly, parts of our NES are designed for fairly slow FPGA. Even though faster FPGA's are eventually available, these portions have not been re-designed to reduce their pipeline depths. Based on manual examination of the Verilog RTL code, we expect the approximately 130 pclk (35 NES clock) delay through the current NES to be reduced to about 88 pclk (22 NES clock) if a re-implementation is done.

5.2.3 Processor Overhead

We now examine the processor overhead for the various message types. The processor overhead is related to but not exactly the same as the aP software components of the latency path – the former measures occupancy, while the latter measures critical path.

Table 5.5 reports the processor overheads incurred for minimum size messages. More detailed breakdowns for these numbers are shown in Tables 5.6 and 5.7. The processor overhead for messages with sizes ranging from 2 to 20 words are reported in Figure 5-2; this shows a significantly higher processor overhead increment when the message size increase crosses a cache-line boundary.

The processor overhead for Express message is significantly lower than those for the other message types. This is especially true for message transmit. This is achieved

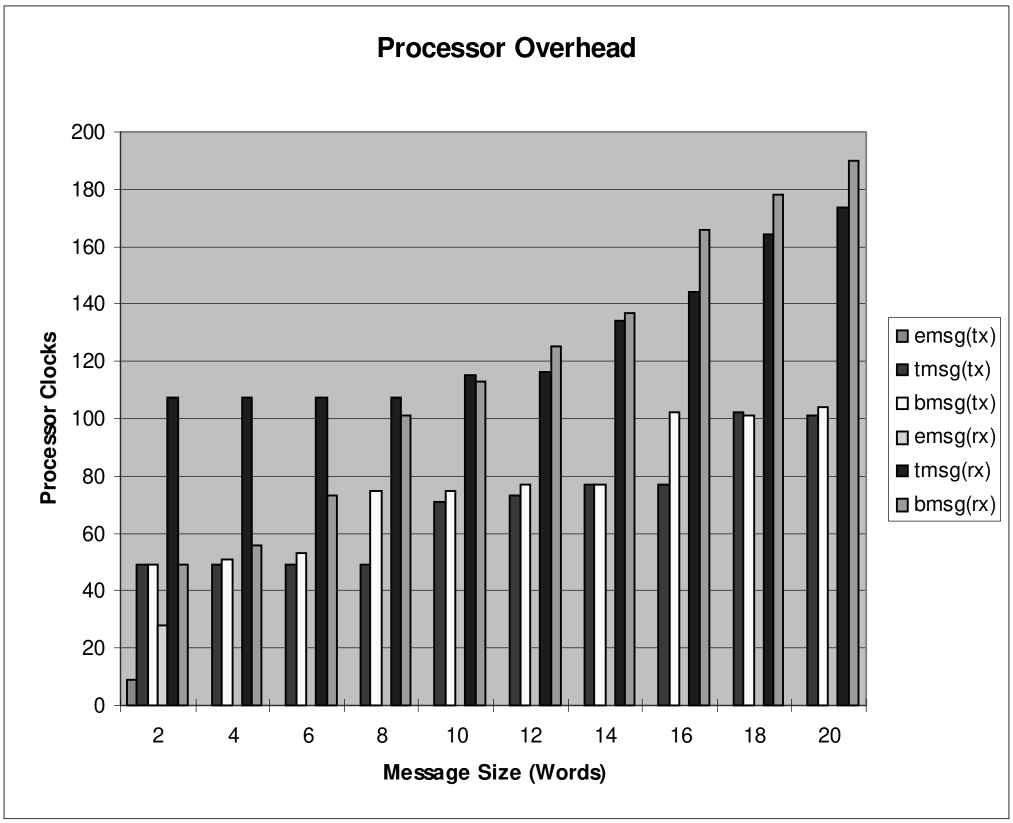


Figure 5-2: Message transmit (tx) and receive (rx) processor overhead for Express (emsg), Express-TagOn (tmsg) and Basic (bmsg) messages as message size varies from 2 to 20 (4 byte) words.

Processor overhead components	Express (proc clks)	Express-TagOn (proc clks)	Basic (proc clks)
Data copying (active cycles)	2	2	2
Data copying (stalled cycles)	0	30	20
Book-keeping	7	41	27
Total	9	73	49

Table 5.6: Breakdown of processor overhead for sending minimum size Express, Express-TagOn and Basic messages.

Processor overhead components	Express (proc clks)	Express-TagOn (proc clks)	Basic (proc clks)
Reading data (active cycles)	2	3	1
Reading data (stalled cycles)	26	56	28
Book-keeping	0	57	20
Total	28	116	49

Table 5.7: Breakdown of processor overhead for receiving minimum size Express, Express-TagOn and Basic messages.

by bundling all the off-chip access into an uncached transaction, so that off-chip access cost is minimized. As reflected in Table 5.6, off-chip access cost, which shows up as stalled processor cycles caused by cache-miss and cache push-out, dominates the processor overhead for the other message types.

Processor overhead incurred during message receive is again dominated by off-chip access, particularly the latency of reading from the NES. It is very difficult to reduce this read latency which shows up as processor stall cycles unless pre-fetching is done. Though a good idea, pre-fetching is not easy when message receive code is integrated into real programs; as such, our micro-benchmarks do not include pre-fetching. Among the different message types, Express message still manages to achieve the lowest processor overhead because it does not require any book-keeping by the aP software.

Next, we present statistics on the processor overhead of sending multi-cast messages under Basic and Express-TagOn message passing mechanisms. The overhead is significantly lower with Express-TagOn as the number of multi-cast destinations increases, since the aP specifies the content of the multi-cast message to the NES only once. For a payload of 80 bytes, Basic message incurs an overhead of 104 processor clocks (0.74 μs @ 140 MHz pclk) per destination. Using Express-TagOn, the cost for the first destination is 96 processor clocks (0.68 μs), but each additional destination incurs only 10 processor clocks (0.07 μs).

Finally, if multiple threads share a message queue, Basic message will require the threads to coordinate their usage using mutex locks. This adds processor overhead. To get an idea of this cost, we implemented user-level mutex locks with the load-reserve and store-conditional pair of atomicity instructions available on PowerPC processors and measured the cost on a real system. Even in the case where there is no contention, it costs about 30 processor clocks to take or to release a lock. The high cost is partly due to serialization of execution within the PowerPC 604e processor, and partly due to the bus operations triggered by these instructions. Bracketing a message send or receive code sequence with a lock acquire and release will therefore add about 60 processor clocks of processor overhead. In contrast, the uncached write to send an Express or Express-TagOn message is inherently atomic and hence no locking is needed. Atomicity on the receive side can similarly be achieved using 64bit uncached read.

Sharing a Basic Message queue among concurrent threads adds not only the overhead of using mutex locks to coordinate message queue usage, but also introduces thread scheduling headaches. A thread that has obtained the right to use a message queue will block other threads if it is suspended, say due to page fault, before completing usage of the message queue.

5.3 Multiple Message Queues Support

This section provides quantitative evaluation of the virtual message queues idea as implemented with our Resident/Non-resident strategy. First, the impact of message queue virtualization and supporting a moderate number of hardware queues is quantified. We show that the added latency from queue name translation, and the slightly more complex logic is immaterial – it is very small compared to the overall message passing latency today.

Next, the performance of Non-resident Basic message is presented and compared to Resident Basic message performance. The evaluation shows that the latency of Non-resident implementation is at worst five times longer than Resident implementation, while bandwidth is reduced by two thirds. This level of performance is perfectly adequate performance as a backup, although it is not as good as we had anticipated. The message send portion of the Basic message interface presents many challenges to efficient emulation by the sP; other message send interfaces, notably that of Express and Express-TagOn messages are more conducive to sP emulation.

Despite not living up to our expectations, the Non-resident Basic message performance is still very respectable. The one-way latency of 12 μs is not unlike those reported for Myrinet. Myrinet performance varies depending on the host system used and the interface firmware running on the Lanai, its custom designed embedded processor. Numbers reported by various researchers indicate one-way latency as low as 5.5us. However, well known user-level interfaces, such as AM-II and VIA incur one-way latency numbers of 10.5us and 30us respectively [101].

Although the Non-resident bandwidth of around 18 MByte/s may look low, this is achieved with the sP performing send and receive of messages with only 80 byte payload. If all that we care about is how fast sP can implement block transfers, other methods presented in Section 5.4 achieve bandwidths of over 60 MByte/s.

An insight we gained from this evaluation is the importance of handling the most common communication operations completely in hardware instead of involving firmware. The relatively long latency of many well known message passing NIU's/machines, such as Myrinet and SP-2, is due in large part to using firmware to handle *every* message send and receive operation. It is common to attribute their long latency to their location on the I/O bus. While that is partly responsible, firmware is by far the biggest culprit.

Component	Latency Penalty (NES clks)
TxQ state load	1
Destination translation	3
TxQ state write-back	0
RQID lookup	1
RxQ state load	1
RxQ state write-back	0

Table 5.8: The latency penalty incurred by Resident message queues in order to support multiple virtual message queues. The message queue state write-back penalties are zero because they are not on the latency critical path.

5.3.1 Performance Cost to Resident Message Queues

Message queue virtualization, as implemented in StarT-Voyager, imposes two types of performance penalty on hardware message queues: (i) the cost of supporting a moderately large number of hardware message queues; and (ii) the cost of queue name translation and matching virtual queue names to hardware queues. Latency is the performance metric most affected, incurring a total overhead of six NES clock cycles (24 processor cycles). This is about 10% of Express message latency, which is the shortest among all message types. Table 5.8 shows the breakdown for this overhead.

The transmit and receive queue state loading overhead is due to the way the NES micro-architecture implements multiple hardware message queues. As described in Section 4.5.1, message queue states are separated from the logic that operates on them. The former are grouped into queue state files, akin to register files, while the latter is structured as transmit and receive functional units. This organization achieves more efficient silicon usage as the number of message queues increases since the functional units need not be replicated. With this organization, queue state has to be loaded into the functional units and subsequently written back. Each load or write-back operation takes one cycle. Its latency impact is minimal and bandwidth impact is practically zero since it is hidden by pipelining.

Virtualization of queue names requires message destination translation during transmit and RQID (Receive Queue IDentity) tag lookup to de-multiplex incoming packets. Similar to cache-tag matching, the RQID tag lookup incurs a latency penalty of one NES clock cycle. It has no bandwidth impact as the lookup is done with dedicated CAM (Content Addressable Memory) in the NESCtrl chip.

Message destination translation adds three NES cycles to latency: one cycle to generate the address of the translation table entry, another to read the translation information itself, and a third to splice this information into the out-going packet. Because the NES design places the translation information in the same SRAM bank as message queue buffers, reading the translation information has a bandwidth impact. SRAM port usage, originally $\lceil(n/8)\rceil$ cycles for an n-byte message, is lengthened by one cycle. This only has an impact on very small messages. Furthermore, an implementation can use a separate translation table RAM to avoid this bandwidth penalty altogether.

The latency penalty of six NES clock cycles is so low that it will remain a small part of the overall latency even if better implementation technology is used for the NES and the network so that their latency contribution is greatly reduced. We can make an argument that the six cycle overhead value is small relative to the latency of a system bus transaction, which is at least four bus clocks today and unlikely to decrease. As long as the NIU is external to the processor, any message passing communication will require at least two system bus transactions, and probably more.

5.3.2 Comparison of Resident and Non-resident Basic Message Queues

This section examines the performance of Non-resident Basic Message queues implemented as described in Section 4.6.2. We present bandwidth and latency numbers for the four combinations resulting from the cross-product of the sender and the receiver using either Resident or Non-resident queues. Processor overhead numbers are only minimally increased by the switch from Resident to Non-resident queues, and are not

TxQ Resident?	RxQ Resident?	Bandwidth (MByte/s)
Yes	Yes	53.15
Yes	No	25.75
No	Yes	17.80
No	No	17.40

Table 5.9: Bandwidth achieved with Basic Message when transferring 4 kByte of data from one node to another. The four cases employ different combinations of Resident and Non-resident queues at the sender and receiver.

presented here. The marginal increase is due to longer latency of accessing DRAM.

Table 5.9 lists the bandwidth achieved for 4-kByte block data transfers. When both sender and receiver use Non-resident message queues, the bandwidth is a third of the Resident implementation's bandwidth. The same table also shows that the bottleneck is at the sender's end, since close to half the Resident queues bandwidth is attained if the sender uses a Resident queue while the receiver uses a Non-resident queue.

Figure 5-3 reports one-way message latency under the four combinations of sender and receiver queue types. The bar graph displays the absolute latency numbers, while the line graphs join the points indicating the latency "slow-down" ratios, *i.e.* the ratio of the (longer) latencies incurred with Non-resident queues compared to the corresponding latencies under Resident queues. The latency deterioration is worse for smaller messages because several Non-resident queue overhead components are fixed regardless of message size. The figure also shows that Non-resident message transmission causes greater latency deterioration than Non-resident message receive. In the worst case of a one word (4 Byte) Basic message sent between Non-resident queues, latency is almost five times longer.

Accounting for sP Emulation Costs

To facilitate understanding of sP emulation cost, we differentiate between two cost categories. The first is the inherent cost of sP firmware processing. This is determined by our NES micro-architecture. The second is the Basic message interface itself;

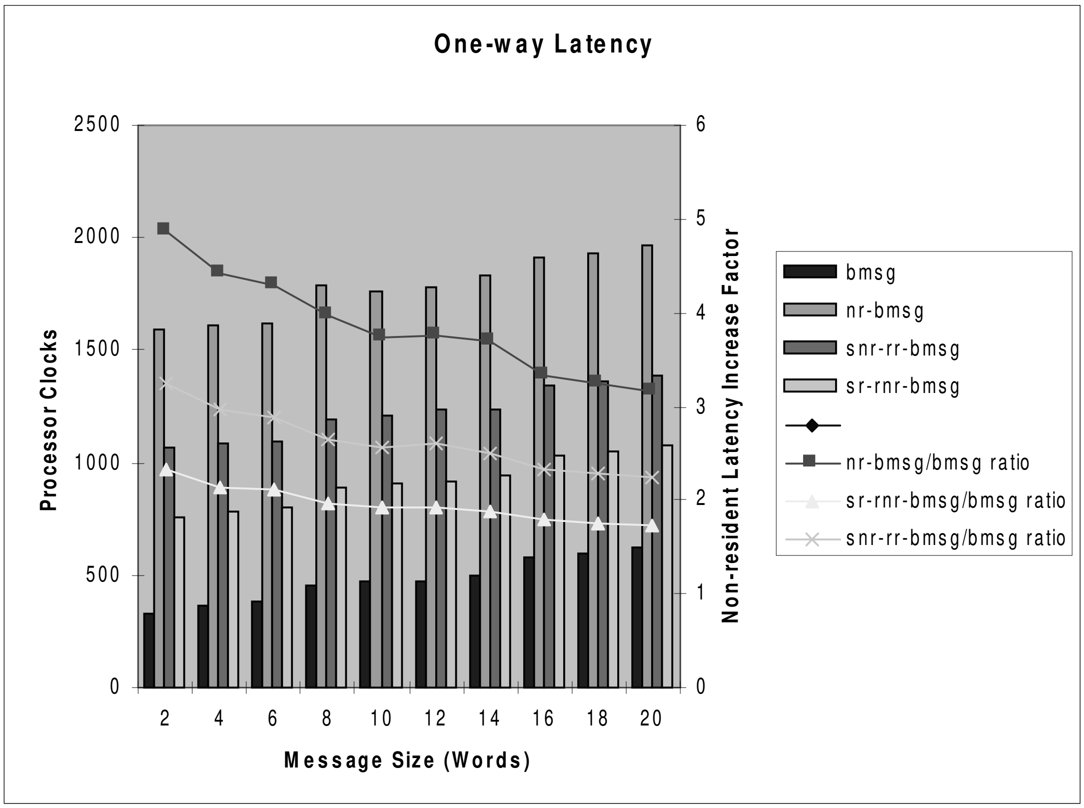


Figure 5-3: One-way latency of Basic Message when sent from and received in different combinations of Resident and Non-resident Basic Message queues. The "bmsg" case is between Resident transmit and Resident receive queues. The "nr-bmsg" case is between Non-resident transmit and Non-resident receive queues. The "snr-rr-bmsg" case is between a Non-resident sender and a Resident receiver, while the "sr-rnr-bmsg" case is between a Resident sender and a Non-resident receiver.

obviously, different interface design choices influence the efficiency of sP emulation.

We defer detailed discussions of the efficiency of sP firmware processing to Section 5.4. It suffices to say that sP handling of each event imposes fixed overheads which can be significant if events are frequent. Multi-phase processing of an operation exacerbates this cost because state has to be saved and restored between phases.

The Basic message interface design is difficult to implement efficiently with sP code for two reasons: (i) information is exchanged in a way that forces the sP to process a message transmission in two phases; (ii) our choice of buffer queue organization with a dynamically varied end of queue. Details follow.

Updating a transmit queue producer pointer triggers the first phase of sP processing. The sP is, however, unable to complete the emulated transmit because two pieces of message header information – message destination and size – have to be read from the SMP's main memory. Because the sP cannot directly read the aP's DRAM, it has to first marshal the message header and data into NES SRAM. While waiting for this to complete, it must continue to service other requests to avoid possible deadlocks. This incurs context switching overhead, including storing aside information for the second phase, which then has to locate it.

Basic message receive also incurs multiple phases of sP processing, but fortunately, the second phase can be aggregated, and there is little state carried from the first phase to the second one. The main task of the latter is to free up message buffer space in the proxy receive queue[3]. In retrospect, we could have extended the command set of the local command queue to include one that frees up hardware queue space. This, together with the FIFO property of the local command queue would make it possible for the sP to completely process a Basic message receive in a single phase.

Not all Non-resident transmit emulation is as expensive as that for Basic message. The transmit part of Express and Express-TagOn interface is in fact much more conducive to sP Non-resident implementation because message destination and size information are available in the bus transaction that triggers sP processing. This

[3]The proxy queue is the hardware queue onto which packets from multiple emulated queues are multiplexed.

enables the sP to emulate message transmit in a single phase. As a result of this simplification, the transmit sP occupancy is expected to be half that of Basic message.

Implementing Basic message Non-resident transmit is also particularly difficult due to a seemingly innocuous buffer queue design decision. Basic message queues are circular queues with variable size buffers. While having many advantages discussed in Section 3.3.1, variable buffer size leads to the possibility of a buffer at the end of a message queue's linear address region straddling both the end, and the beginning of the queue's address range. This situation is inconvenient for software; for example, it makes it impossible to reference a message buffer as a "struct" in C programs.

To handle this problem, we made the effective end of a queue variable – as long as the space left at the end is insufficient for the maximum size message, the queue is wrapped back to the beginning. The amount of space skipped over is dependent on the dynamic composition of buffer sizes encountered. This was a bad decision because of problems it causes and because there is an easier solution.

Our buffer queue structure makes buffer allocation code tedious. This inconvenience and inefficiency is greatly compounded in Non-resident transmit because the dynamically varied end of a queue shows up in *both* the emulated Non-resident queue, *and* the hardware proxy queue. Furthermore, the dynamic end of an emulated queue cannot be determined until the second phase processing.

The straight forward way to deal with this complexity is to stall processing whenever any queue, either proxy or emulated, approaches its dynamically determined end. Processing only resumes when message size information available during the second phase resolves the precise point at which the queue has reached its end.

A second approach avoids stalling by having phase one processing marshal sufficient data into two different SRAM locations. Only one of these two copies is used during phase two, depending on where the queue wraps back to its beginning. Phase two processing ignores redundant data by sending loop-back "garbage" packets which it discards. This is the implementation we chose.

The complexity of this variable end of queue design could have been avoided with an alternate solution that permits variable packet sizes without breaking any packet into non-contiguous address region. This solution imposes the restriction that message queues have to be allocated in page-size granularity. When the physical pages are mapped into an application's virtual address space, an extra virtual page is used at the end, which maps to the first physical page, *i.e.* both the first and last virtual pages of a message queue map onto the same physical page. This allows a packet occupying non-contiguous physical addresses to end up in contiguous virtual addresses.

5.4 Performance Limits of the sP

This section examines the efficacy of the sP in the StarT-Voyager NES design. As described in Chapter 4, the NES Core provides the sP with a large, flexible set of functions, leaving few doubts that the sP is functionally capable of emulating almost any communication abstraction. What is less apparent is the performance the sP can achieve, and the factors limiting this performance.

The results reported in this section show that when driven to the limit, sP performance is constrained by either context switching or off-chip access. When the amount of work involved in each sP invocation is small, the context switch overhead – event poll, dispatch and book-keeping (especially for multi-phase operations) – dominates. When each sP invocation orchestrates a large amount of data communication, the cost of sP off-chip access dominates.

This result validates several of our concerns when designing the sP/NES Core interface. These include: (i) crafting an interface which minimizes the number of phases in sP processing of an event, and (ii) reducing the number of sP off-chip accesses through mechanisms like One-poll. On the other hand, it also reveals one area of sP function that we did not consider carefully during our design – the burden of NES Core hardware resource allocation and deallocation.

At a higher level, the results are a reminder that the sP's performance is ultimately lower than that of dedicated hardware. Therefore, it is crucial that it is not invoked too often. If the sP's duties are limited to a small number of simple tasks, its processing cost can potentially be lowered because of reduced book-keeping. In our opinion, an sP so constrained is better replaced with FPGA's.

Two sets of experiments contribute to the results of this section. The first is the Non-resident Basic message performance first presented in Section 5.3.2. We revisit the performance results, dissecting it to quantify the cost of generic sP functions. Details are reported in Section 5.4.1.

In the second set of experiments, the sP implements block transfer in several ways. These experiments differ qualitatively from the first set in that a fair amount of communication is involved each time the sP is invoked. Because the sP overhead from dispatch and resource allocation is amortized, these experiments reveal a different set of performance constrains. Details are reported in Section 5.4.2.

5.4.1 sP Handling of Micro-operations

We instrumented our simulator and the Non-resident Basic message performance benchmarks to obtain the cost for each invocation of the sP when it emulated Basic message queues. These numbers are tabulated in Table 5.10. To double check that these numbers are reasonable, we also inferred from the bandwidth numbers in Table 5.9 that at the bottleneck point, each Non-resident Basic packet takes 660 processor clocks and 457 processor clocks for transmit and receive respectively[4]. These numbers are compatible with those in Table 5.9. They also show that sP occupancy constrains transmit bandwidth, but not the receive bandwidth.

Why does the sP take several hundred processor clocks to process each of these events? Firstly, all our code is written in C and then compiled with GCC. Manual examination of the code suggests that careful assembly coding should improve performance by at least 20-30%. Secondly, the sP code is written to handle a large number of events. Polling and then dispatching to these events, and saving and restoring the

[4]Each emulated Basic Message packet carries 84 bytes of data. When sP emulates message transmit, it achieves 17.8 MBytes/s, *i.e.* it processes 212 thousand packets every second. Since the sP operates at 140 MHz, this works out to be one packet every 660 cycles. Similarly, when the sP emulates message receive, it achieves 25.75 MBytes/s, *i.e.* it processes 306.5 thousand packets every second. This works out to be one packet every 457 cycles.

Component	sP occupancy (proc clks/packet)
Tx emulation, Phase 1 (Marshal data)	346
Tx emulation, Phase 2 (Destination trans and launch)	281
Total	627
Rx emulation, Phase 1 (Demultiplex data)	209
Rx emulation, Phase 2 (Free buffer and update queue state)	112
sP free receive buffer	59
Total	380

Table 5.10: Breakdown of sP occupany when it implements Non-resident Basic message.

state of suspended, multi-phase processing all contribute to the cost.

As illustration, we obtained the timing for sP code fragments taken from the Non-resident Basic message implementation. The sP's top-level dispatch code using a C switch-case statement takes 13 processor clocks. When this is followed by dequeuing the state of a suspended operation from a FIFO queue, and then a second dispatch, an additional 35 processor clocks is incurred.

Hardware resource management, such as allocation and deallocation of space in the local command queues, also incurs overhead. With each task taking several tens of cycles, these dispatches, lookups and resource management very quickly add up to a large number of sP processor cycles.

5.4.2 sP Handling of Macro-operations

Table 5.11 shows the bandwidth achieved for 4 kBytes block transfer using three transfer methods, two of which involves the sP.

Benchmark Details

The first method uses the NES hardware DMA capability. This has been reported

Transfer Method	Bandwidth (MBytes/s)
NES Hardware DMA	84.40
sP sends and receives	62.41
sP sends, NES hardware receives	70.85

Table 5.11: (4 kByte) Block transfer bandwidth under different transfer methods on StarT-Voyager as measured on the StarT-sim simulator.

earlier but is repeated here for easy comparison. In the second method, the sP is involved at both the sending and receiving ends. The sP packetizes and sends data by issuing aP bus operations to its local command queue to read data into the NES. These are followed by Express-TagOn commands to ship the data across the network. The sP takes advantage of the local command queue's FIFO guarantee to avoid a second phase processing of the transmit packets.

On the receive end, the Express-TagOn receive queue is set up so that the TagOn part of the packet goes into aSRAM, while the header/Express-like part goes into sSRAM. The sP examines only the latter, and then issues aP bus operation commands to its local command queue to move the TagOn data into the appropriate aP DRAM locations. Processing at the receiver sP has a second phase to de-allocate TagOn data buffer space in the receive queue. This can be aggregated, and the reported numbers are from code that aggregates the buffer free action of two packet into one sP invocation.

In the third benchmark, the sP is only involved at the sender end. On the receiving side, the packets are processed by the NES's remote command queue, the same hardware used in NES hardware DMA. The purpose of this example is to characterize sP packetize and send performance. When the sP is responsible for both sending and receiving, the receive process is the likely performance bottleneck because it has two phases. This suspicion is confirmed by the results in Table 5.11 which shows this third transfer method achieving higher performance than the second one.

Result Analysis

The numbers in Table 5.11 can be explained by the conjecture that the limiting factor on sP performance is the number of times it makes off-chip accesses. For example, under the second transfer method, the 62.41 MBytes/s bandwidth implies that the bottleneck processes one packet every 143.5 processor clock (36 bus clocks). As shown in the following table, when the sP sends block data, off-chip access occupies the sP bus for 34 bus clocks per packet. (Short bus transactions involve smaller amounts of data and each occupies the bus for only 4 bus clocks; more data is involved in long bus transactions, each occupying the bus for 5 bus clocks.)

Operation	num & type of bus transaction	bus clocks
Poll for event to handle	2 short	8
3 local command queue commands	3 short	12
1 cache-line write-miss	1 long	5
1 cache-line flush	1 long	5
Poll for command ack	2 short/2 (aggregated)	4
Total		34

Chapter 6

Conclusions and Future Work

This piece of research centers around the thesis that a cluster system NIU should support multiple communication interfaces layered on a virtual message queues substrate in order to streamline data movement both within each node as well as between nodes. To validate this thesis, we undertook the design and implementation of the StarT-Voyager NES, an NIU that embodies these ideas. Our work encompasses design specification, Verilog coding, synthesis and performance tuning, simulator building, functional verification, PC board netlist generation, sP firmware coding and finally micro-benchmark based evaluation. Through this exercise, we obtained a qualitative idea of the design complexity and a quantitative picture of the hardware (silicon) size. Neither issues present any significant impediment to realizing our design.

6.1 What We Did

To enable an NIU to support multiple interfaces, we solved a series of problems, such as sharing a fast system area network in a safe manner without imposing an unreasonable performance penalty, and managing the hardware complexity and cost of supporting multiple interfaces. We introduced a three-layer NIU architecture, described in Chapter 3, to decouple the different issues encountered in this design. Specifically, network sharing protection is handled in the Virtual Queues layer so that the Application Interface layer can devote itself to crafting efficient communication interfaces.

In the Virtual Queues layer, we designed a protection scheme that is both very flexible and cheap to implement. Flexibility is achieved by leaving policy decisions to system software. Implementation is cheap because the scheme requires only simple support from the NIU hardware. We implemented these virtual queues in the StarT-Voyager NES with a combination of hardware Resident queues and firmware emulated Non-resident queues. The former are employed as caches of the latter under firmware control. This design illustrates the synergy between custom hardware functional blocks and the embedded processor in our hybrid NIU micro-architecture.

In the Applications Interface layer, we designed a number of message passing mechanisms catering to messages of different sizes and communication patterns. We also provided sufficient hooks in the NES for firmware to implement cache-coherent distributed shared memory. These hooks allow firmware to participate in and control the outcome of snoopy bus transactions on the computation node, ensuring tight integration into the node's memory hierarchy. Though originally intended for cache-coherent shared memory implementation, these capabilities are used to implement other communication interfaces, such as our Non-resident message queues. The design shows that by building on a basic set of hardware mechanisms, multiple interfaces can be supported with minimal to no per-interface enhancements.

At the micro-architectural level, we examined different options for introducing programmability into the NIU. We considered using one of the node processors but found serious dangers of deadlock and likely performance problems. We finally picked using a dedicated, off-the-shelf embedded processor in the NIU as the most expedient choice. To overcome some performance and functional limitations of this approach, we treated the custom NIU hardware as a coprocessor to the embedded processor, and structured this interface as a set of in-order command and completion queues.

6.2 What We Learned

The merits of our ideas are evaluated on a system simulator we developed. The results show that the Resident/Non-resident implementation of virtual queues is a good idea that achieves the seemingly conflicting goals of high performance, low cost, and flexible functionality. The average performance, expected to be dominated by Resident queue performance, is little worse than if the had NIU supported only a small fixed number of queues. NIU hardware cost is kept low as the large number of queues is implemented in firmware using storage in main memory.

Micro-benchmark based evaluation also illustrates the performance merit of multiple interface support – in StarT-Voyager NES, the best message passing mechanism for a specific communication depends on its characteristics such as data size or communication pattern. The advantage comes from using these characteristics to pick the best data and control path from among the myriad options in today's complex memory hierarchy. It is also significant that despite the generality and flexibility of the StarT-Voyager NES, each mechanism it offers is competitive against implementations that provide only that particular mechanism.

Finally, we evaluated the utility of the off-the-shelf embedded processor in the NIU. While functionally extremely flexible, the embedded processor offers lower performance than custom hardware. We showed that with our design, the embedded processor is limited by either context switch overhead in the case of fine-grain communication events, or by off-chip access when it handles coarse-grain communication events.

While the StarT-Voyager NES is an interesting academic prototype, its implementation is not very aggressive, resulting in compromised performance. If financial and man-power resources permitted, the entire NES except for the embedded processor and its memory system should be integrated into one ASIC. This will improve communication performance, particularly latency. If further resources are available, a generic programmable core integrated into this ASIC could overcome the off-chip access limitation of our embedded processor.

6.3 Future Work

This work addresses the question: "Is a multi-interface NIU feasible?" This question has been answered in the affirmative through a complete design and micro-benchmark performance evaluation of the message passing substrate. This work is, however, just an initial investigation and many open issues remain; we list some below.

Further Evaluation with Real Workload

This thesis focuses more on design, both abstract and concrete, and less on evaluation. Further evaluation of the StarT-Voyager NES with real applications and workload will lead to more definitive answers to the following important questions.

- What are the effects of multiple message passing mechanisms on overall application and workload performance?
- Does more flexible job scheduling enabled by our network sharing model actually improve system utilization?
- What is the performance of Reactive Flow-control when used on real programs under real workload conditions?
- Is our choice of supporting the queues-of-network model and dynamic receive queue buffer allocation necessary? In particular, is either the channel model, or having the message sender specify destination address sufficiently convenient interfaces?

While seemingly simple, these questions can only be answered after the entire system – hardware, system software, and application software – is in place.

Cache-coherent Global Shared Memory

Shared memory is not a focus of this research, but to fully validate the concept of a multi-function NIU, we must demonstrate that this architecture does not penalize shared memory performance in any significant way. The goal is to demonstrate shared memory performance on par with if not better than that delivered by shared memory machines.

Although the current NES functionally supports cache-coherent shared memory, and will deliver good performance when inter-node cache miss rate is low, there are concerns about inter-node cache-miss processing latency and throughput because the sP is always involved in such cases.

Further work needed in this area includes a combination of performance study, scrutiny of and modification to the NES micro-architecture to ensure that all cache-miss latency critical paths and throughput bottlenecks are handled in NES hardware. Some possibilities are to add by-passes, aggressively avoid store-and-forward in the processing paths, and give shared memory traffic priority in resource arbitration. The StarT-Voyager NES, with its malleable FPGA-based hardware, provides a good platform for conducting this piece of research.

Multi-interface NIU vs aP Emulation on Shared Memory NIU

This thesis investigated multi-interface NIU's, *i.e.* NIU's designed to directly support multiple communication interfaces. An increasingly common approach to satisfy application demand for different communication abstractions is to implement fast coherent shared memory support in hardware, and use aP code to emulate all other communication interfaces. Although we believe that this latter approach has both performance and fault-isolation draw-backs due to lack of direct control over data movement, this thesis did not attempt a definitive comparison of the two approaches. This is pre-mature until the implementation feasibility of the multi-interface NIU approach is demonstrated. With this thesis work and further work on the shared memory aspects of the multi-interface NIU architecture as the foundation, a comparative study on how best to support multiple communication interfaces is in order.

Bibliography

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocesors*. Kluwer Academic Publishers, 1991.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro*, 13(3), May/Jun 1993.

[3] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152 – 184, 1995.

[4] M. S. Allen, M. Alexander, C. Wright, and J. Chang. Designing the PowerPC 60X Bus. *IEEE Micro*, 14(5):42 – 51, Sep/Oct 1994.

[5] R. Alverson, D. Callahan, D. Cummings, K. B., A. Porterfiled, and B. Smith. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing*, June 1990.

[6] C. Anderson and J.-L. Baer. Two Techniques for Improving Performance on Bus-based Multiprocessors. In *Proceedings of the First International Symposium on High-Performance Computer Architecture, Raleigh, NC*, pages 264 – 275, Jan. 1995.

[7] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, Nov. 1986.

[8] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON '93 Conference*, Feb. 1993.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*, pages 207–216, July 1995.

[10] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[11] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–36, Jan/Feb 1995.

[12] G. A. Boughton. Arctic Routing Chip. In *Proceedings of Hot Interconnects II, Stanford, CA*, pages 164 – 173, Aug. 1994.

[13] G. A. Boughton. Arctic Switch Fabric. In *Proceedings of the 1997 Parallel Computing, Routing, and Communication Workshop, Atlanta, GA*, June 1997.

[14] E. A. Brewer and F. T. Chong. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proceedings of Symposium on Parallal Algorithms and Architectures '95, Santa Barbara, CA*, 1995.

[15] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilks. An Implementation of the Hamlyn Sender-managed Interface Architecture. In *Proceedings*

of the Second Symposium on Operating Systems Design and Implementation, Seattle, WA., pages 245–259, Oct. 1996.

[16] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilks. Hamlyn: a High-performance Network Interface with Sender-based Memory Management. Technical Report HPL-95-86, Computer Systems Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA., July 1995.

[17] D. Chaiken and A. Agarwal. Software-Extended Coherenct Shared Memory: Performance and Cost. In *??*, page ??, ?? ??

[18] K. Charachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Conference on Computer Architecture*, pages 15 – 26, May 1990.

[19] D. Chiou, B. S. Ang, R. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. E. Hicks, and A. Boughton. StarT-NG: Delivering Seamless Parallel Computing. In *Proceedings of the First International EURO-PAR Conference, Stockholm, Sweden*, pages 101 – 116, Aug. 1995.

[20] Chip Express Corporation, 2323 Owen Street, Santa Clara, CA 95054. *CX Technology Design Manual/1997*, May 1997. (CX2000/CX2001 Family Laser Programmable Gate Arrays).

[21] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verfication of the futurebus+cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, Apr. 1993.

[22] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subrmonian, and T. von Eicken. LogP: Torwards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on*

Principles and Practice of Parallel Programming, San Diego, pages 1 – 12, May 1993.

[23] F. Dahlgren, M. Dubois, and P. Stenstrom. Combined Performance Grains of Simple Cache Protocol Extensions. In *Proceedings of the 21st International Symposium on Computer Architecture*, Apr. 1994.

[24] W. J. Dally, R. Davison, J. A. S. Fiske, G. Fyler, J. S. Keen, R. A. Lethin, M. Noakes, and P. R. Nuth. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, 12(2), Mar/Apr 1992.

[25] W. J. Dally, L. R. Dennison, D. Harris, K. Kan, and T. Xanthopoulos. Architecture and Implementation of the Reliable Router. In *Proceedings of the Hot Interconnects II*, pages 122–133, Aug. 1994.

[26] D. L. Dill. The Murphi Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification, New Brunswich, NJ*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer, July 1996.

[27] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A Message Passing Standard for MPP and Workstations. *Communications of the ACM*, 9(36):84–90, July 1996.

[28] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66 – 76, Mar/Apr 1998.

[29] B. Falsafi and D. A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 229–240, June 1997.

[30] M. Fillo, S. W. Keckler, W. J. Dally, et al. The M-Machine Multicomputer. *International Journal of Parallel Programming*, 25(3):183 – 212, June 1997.

[31] M. P. I. Forum. Document for a Standard Message-passing Interface. Technical Report CS-93-214, University of Tennessee, Nov. 1993.

[32] M. Frigo and V. Luchangco. Computation-Centric Memory Models. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 240–249, June 1998.

[33] M. Galles. Spider: A High-Speed Network Interconnect. *IEEE Micro*, 17(1):34 – 39, Jan/Feb 1997.

[34] G. R. Gao and V. Sarkar. Location Consistency: Stepping Beyond the Memory Coherence Barrier. In *Proceedings of the 24th International Conference on Parallel Processing, Oconomowoc, Wisconsin*, Aug. 1995.

[35] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[36] R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, Jan/Feb 1996.

[37] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64 – 75, 1989.

[38] F. Hady, R. Minnich, and D. Burns. The Memory Intergrated Network Interface. In *Proceedings of the Hot Interconnects II*, pages 10–22, Aug. 1994.

[39] E. Hegersten, A. Saulsbury, and A. Landin. Simple COMA Node Implementation. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, Jan. 1994.

[40] J. Heinlein. *Optimized Multiprocessor Communication and Synchronization Using a Programmable Protocol Engine*. PhD thesis, Stanford University, Stanford, CA, Mar. 1998.

[41] J. C. Hoe. Effective Parallel Computation on Workstation Cluster with User-level Communication Network. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb. 1994.

[42] J. C. Hoe. StarT-X: A One-Man-Year Exercise in Network Interface Engineering. In *Proceedings of Hot Interconnects VI*, pages 137–146, Aug. 1998.

[43] J. C. Hoe and M. Ehrlich. StarT-Jr: A parallel system from commodity technology. In *Proceedings of the 7th Transputer/Occam International Conference*, Nov. 1996.

[44] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proceedings of Hot Interconnects I*, Aug. 1993.

[45] R. W. Horst and D. Garcia. ServerNet SAN I/O Archtiecture. In *Proceedings of Hot Interconnects V*, 1997.

[46] IBM Microelectronics. *PowrPC 604 RISC Microprocessor User's Manual*, Nov. 1994. (MPC604UMU-01, MPC604UM/AD; also available from Motorola Inc. Semiconductor Products Sector).

[47] IEEE Computer Society. *Scalable Coherent Interface, IEEE Standard 1596-1992*, Aug. 1993.

[48] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: a Bridge Between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[49] Intel Corporation, Intel Corporation, Supercomputer Systems Division, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006. *Paragon XP/S Product Overview*, 1991.

[50] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pages 213 – 228, Dec. 1995.

[51] A. Kagi, D. Burger, and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 170–180, June 1997.

[52] M. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive Snoopy Caching. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 244 – 254, 1986.

[53] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115 – 132, Jan. 1994.

[54] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13 – 21, May 1992.

[55] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: a New Dimension for Cray Research. In *Digest of Papers, COMPCON Spring 93, San Francisco, CA*, pages 176 – 182, Feb. 1993.

[56] L. I. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *Proceedings of the First International Symposium on High-Performance Computer Architecture, Raleigh, NC*, pages 286–297, Jan. 1995.

[57] J. Kubiatowicz, D. Chaiken, A. Agarwal, A. Altman, J. Babb, D. Kranz, B.-H. Lim, K. Mackenzie, J. Piscitello, and D. Yeung. The Alewife CMMU: Addressing the Multiprocessor Communications Gap. In *HotChips VI*, pages 31 – 42, Aug. 1994.

[58] J. D. Kubiatowicz. *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*. PhD thesis, Massachusetts Institute of Technology, 1998.

[59] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chaplin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, Il*, pages 302 – 313, Apr. 1994.

[60] J. S. Kuskin. *The FLASH Multiprocessor: Designing a Flexible and Scalable System*. PhD thesis, Stanford University, Stanford, CA, Nov. 1997. (CSL-TR-97-744).

[61] L. Lamport. How to Make a Multiprocessor Computer Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[62] J. Landou and D. Lenoski. The SGI Origin: A CC NUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241 – 251, June 1997.

[63] C. E. Leiserson. Fat-trees: Universal Networks for Hardware-efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10), Oct 1985.

[64] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[65] D. Lenoski, J. Laudon, K. Charachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocesor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148 – 159, May 1990.

[66] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 92 – 103, 1992.

[67] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 4(7):321 – 359, Nov. 1989.

[68] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 308 – 317, May 1996. (This machine, made by Sequent, was later renamed NUMA-Q.).

[69] K. Mackenzie. *An Efficient Virtual Network Interface in the FUGU Scalable Workstation*. PhD thesis, Massachusetts Institute of Technology, 1998.

[70] K. Mackenzie, J. Kubiatowicz, A. Agarwal, and M. F. Kaashoek. FUGU: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor. Mit/lcs/tm-503, MIT Laboratory for Computer Science, Oct. 1994.

[71] K. Mackenzie, J. Kubiatowicz, M. Frank, V. Lee, A. Agarwal, and F. Kaashoek. Exploiting Two-Case Delivery for Fast Protected Messaging. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[72] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufman Publishers, Inc., San Francisco, CA, second edition, May 1994.

[73] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie-Mellon University, May 1992.

[74] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21 – 65, Feb. 1991.

[75] J. M. Mellor-Crummey and M. L. Scott. Synchronization Without Contention. In *Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 269 – 278, Apr. 1991.

[76] Motorola. *MPC105 PCIB/MC User's Manual*, 1995.

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

[78] S. S. Mukherjee and M. D. Hill. The Impact of Data Transfer and Buffering Alternatives on Network Interface Design. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[79] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint Multiprocessor Simulation, Toolkit for Intel x86 Architectures. In *Proceedings of 1996 International Conference on Computer Design*, Oct. 1996.

[80] R. H. Nikhil, G. M. Papdopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th International Symposium on Computer Architecture Conference Proceeding*, pages 156–167, May 1992.

[81] R. S. Nikhil. Cid: A Parallel "Shared-memory" C for Distributed Memory Machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing, Ithaca, NY*. Springer-Verlag, Aug. 1994.

[82] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. Exploiting Parallelism in Cache Coherency Proctocol Engines. In *Proceedings of the First International EURO-PAR Conference, Stockholm, Sweden*, pages 269–286, Aug. 1995.

[83] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Memory Multiprocessor. In *Proceedings of the International Conference on Parallel Processing '95*, 1995.

[84] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Message (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 2(5):60–73, April-June 1997.

[85] G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle. *T: Integrated Building Blocks for Parallel Computing. In *Proceedings of Supercomputing '93, Portland, Oregon*, pages 624 – 635, Nov. 1993.

[86] S. Park and D. L. Dill. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, June 1996.

[87] S. Park and D. L. Dill. Verification of Cache Coherence Protocols By Aggregation of Distributed Transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.

[88] P. Pierce. The NX/2 Operating System. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume 1 of 2, pages 384 – 390, 1988.

[89] F. Pong. *Symblock State Model: A New Approach for the Verification of Cache Coherence Protocols*. PhD thesis, University of Southern California, Aug. 1995.

[90] F. Pong and M. Dubois. Formal Verification of Delayed Consistency Protocols. In *Proceedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[91] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, Il*, pages 325 – 336, Apr. 1994.

[92] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[93] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An Argument for Simple COMA. In *Proceedings of the First International Symposium on High-Performance Computer Architecture, Raleigh, NC*, pages 276–285, Jan. 1995.

[94] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Cambridge, MA*, pages 174 – 185, Oct. 1996.

[95] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297 – 307, Oct. 1994.

[96] S. 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, Cambridge, MA*, pages 26 – 36, Oct. 1996.

[97] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. CSG Memo 398, MIT Laboratory for Computer Science, Jan. 1998.

[98] X. Shen, Arvind, and L. Rodolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. CSG Memo 414, MIT Laboratory for Computer Science, Jan. 1999.

[99] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium On Computer Architecture, Atlanta*, May 1999.

[100] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the Nineteenth Annual International Symposium on Computer Architecture Conference*, pages 256–266, May 1992.

[101] T. von Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, 31(11):61 – 68, Nov. 1998.

[102] W.-D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. The Synfinity Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 98–107, June 1997. (Original title was The Mercury Interconnect Architecture: a Cost-Effective Infrastructure for High-Performance Servers.).