Fine-Grain Distributed Shared Memory on Clusters of Workstations

by

Ioannis T. Schoinas

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1998

Abstract

Shared memory, one of the most popular models for programming parallel platforms, is becoming ubiquitous both in low-end workstations and high-end servers. With the advent of low-latency networking hardware, clusters of workstations strive to offer the same processing power as high-end servers for a fraction of the cost. In such environments, shared memory has been limited to page-based systems that control access to shared memory using the memory's page protection to implement shared memory coherence protocols. Unfortunately, false sharing and fragmentation problems force such systems to resort to weak consistency shared memory models that complicate the shared memory programming model.

This thesis studies fine-grain distributed shared memory (FGDSM) systems on networks of workstations to support shared memory and it explores the issues involved in the implementation of FGDSM systems on networks of commodity workstations running commodity operating systems. FGDSM systems rely on fine-grain memory access control to selectively restrict reads and writes to cache-block-sized memory regions. The thesis presents Blizzard, a family of FGDSM systems running on a network of workstations. Blizzard supports the Tempest interface that implements shared memory coherence protocols as user-level libraries. Therefore, application-specific protocols can be developed to eliminate the overhead of the fine-grain access control.

First, this thesis investigates techniques to implement fine-grain access control on commodity workstations. It presents four different techniques that require little or no additional hardware (software checks with executable editing, the memory's controller ECC, a combination of these two techniques, and a custom fine-grain access control accelerator board). Furthermore, the thesis examines the integration of hardware fine-grain techniques within commodity operating systems.

Second, this thesis investigates messaging subsystem design for shared memory coherence protocols. It explores implications of extending Berkeley active messages so that explicit polling is not required and protocols are not limited to request/reply semantics. The thesis explores supporting implicit polling using binary rewriting to insert polls in the application code. In addition, it shows that shared memory coherence protocols, while not pure request/reply protocols, have bounded buffer requirements. Accordingly, it proposes a buffer allocation policy that does not require buffers in a node's local memory in the common case, yet is robust enough to handle arbitrary traffic streams.

Third, this thesis investigates extending FGDSM systems to clusters of multiprocessor workstations. FGDSM systems are especially suited for supporting shared memory on multi-

processor networks, because they transparently extend the fine-grain sharing within the node across the network. The thesis identifies the shared FGDSM resources to which access should be controlled in a multiprocessor environment and proposes techniques to address the synchronization issues for each resource that are based on the frequency of accesses to that resource.

Fourth, this thesis investigates address translation in network interfaces to support zerocopy messaging directly to user data structures. Good messaging performance is important for application-specific protocols that push the limits of the messaging hardware. The thesis evaluates a series of designs with increasingly higher operating system support requirements. It proposes techniques to take advantage of locality in the source or destination virtual memory addresses, yet degrade gracefully when the capacity of the address translation structures is exceeded.

Acknowledgments

This journey began six and a half years ago when I first arrived in Madison and enrolled as a graduate student in the Computer Sciences Department. I never got used to Madison's long and cold winters, but looking back through the years, I must say that it has been a rewarding experience. I would like to thank all the people that contributed to the successful conclusion of my journey.

I would like to thank my advisor, Mark Hill, for his guidance, support, and encouragement throughout my work. Mark with Jim Larus and David Wood led the Wisconsin Wind Tunnel project and created the intellectual environment that made my work possible. The members of my thesis committee, Mark Hill, David Wood, Marvin Solomon, Pei Cao and Charles Kime have my gratitude for their patience in reading my thesis and for their critical comments in its content and presentation.

I would like to thank all the members of the Wisconsin Wind Tunnel group and the computer architecture community for their comments and debates. Many thanks to Eric Schnarr, Babak Falsafi, Steven Reinhardt, Rob Pfile, Brian Toonen, Chris Lukas, Mark Dionne, Yuanyuan Zhou, and Rich Martin for their contributions in building Blizzard on Wisconsin COW. I would also like to thank Madhu Talluri, Andy Glew, and Steve Scott for their comments on my work. Thanks to Satish Chandra, Trishul Chilimbi, and Alain Kagi for bravely serving as my guinea pigs (i.e., Blizzard users).

I would like to thank the people in the University of Crete for helping me start on the path that lead me to this accomplishment. Yannis Fragiadakis offered me my first computer job. Manolis Katevenis sparked my interest in computer architecture. Stelios Orfanoudakis and Petros Kofakis guided throughout my graduate work in Crete. Finally, I would like to thank my friends in Salonica and Heraclion for sharing with me the pains and pleasures of growing and for encouranging me on my journey to Madison.

I would like to thank the greek community of UW-Madison, and especially Yannis Ioannidis, Thanos Tsiolis and Yiorgos Kalfas for their warm welcoming. Thanks to my old roommates, Markos Zaharioudakis, Gul Gurkan, Yannis Christou, Andreas Moshovos, and Josef Betouras, for their friendship. I should also thank them along with Stefanos Kaxiras, Afroditi Michailidi, Dionisios Pnevmatikatos and Minos Garofalakis for our coffee breaks. I wish also to thank Bill and Katherine Parks for their support and love ever since the first time we met.

My journey would not have been possible without the love that my family gave to me throughout my life. I would like to thank my father, Theofilos Schoinas, and my mother, Eirini Diakaki-Schoina, for patiently raising me to adulthood and for providing me with a solid education. I cannot ever repay my mother for her contribution in building my character and for

showing me the true meaning of love, devotion, and sacrifice. I cannot thank my brother, Michael Schoinas, enough for taking care of things while I was studying abroad. Finally, I would like to thank the love of my heart, Leah Parks, for her love, support and friendship. If it wasn't for you Leah, Madison's winters would have been colder and lonelier. I still can't believe that I met someone like you in Madison, Wisconsin.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis proposes fine-grain distributed shared memory (FGDSM) systems on clusters of workstations (COWs) to support parallel programs and it explores the issues involved in the design and implementation of FGDSM systems on clusters of commodity workstations running commodity operating systems. FGDSM systems rely on fine-grain access control to selectively restrict memory accesses to cache-block-sized memory regions.

The thesis presents Blizzard, a family of FGDSM systems running on the Wisconsin Cluster Of Workstations. Blizzard supports the Tempest interface [Rei94] that implements shared memory coherence as user-level libraries. In this way, application-specific protocols can be developed to eliminate the overhead of fine-grain access control.

This section is organized as follows. Section 1.1 discusses the context for this work and places Blizzard in perspective to other approaches to support parallel programs. Section 1.2 presents an overview of the COW platform and discusses the structure of Blizzard. Finally, Section 1.3 presents a brief summary of the most important points in the main chapters of the thesis.

1.1 Fine-Grain Distributed Memory In Perspective

One of the most persistent themes in the history of computing has been the effort to harness the power of many processing units to accomplish a single task faster than using a single unit. Unlike sequential computing where the *von Neumann* model [BGvN46] has been dominant right from the start, no consensus has ever been reached on how to organize and control many activities for a single purpose. Numerous parallel programming models have been proposed to

facilitate this goal. Equally numerous parallel programming languages and programming environments have been developed to support specific parallel programming models.

Among parallel programming models, shared memory is popular because it provides a natural extension to the sequential programming model. Another popular programming model is message-passing, which exposes the inherent nature of a parallel platforms as a collection of processing units connected with a network through which they can exchange messages. Other approaches, such as data-parallel models, hide the parallel nature of the machine. They focus on extracting available parallelism using high-level constructs that define operations for vector variables. More exotic schemes such as dataflow models advocated a complete overhaul of the *von Neumann* model in favor of the programmer directly specifying the data dependency graph.

The lack of consensus at the basic level of the programming models is obvious in the history of the parallel computing platforms. System designers diverged to a greater or lesser extent from the *von Neuman* model in an attempt to directly support different parallel programming models. After more than thirty years of exploring alternative avenues, some consensus has been reached on the general form of parallel platforms. System designers have realized that you have to follow the performance curve of the microprocessors very closely and the size of the parallel market is not large enough to amortize the high fixed costs associated with the development of completely new hardware. System designers today agree that parallel machines should use commodity components (e.g., standard microprocessors). There two prevalent design approaches in building parallel platforms out of commodity processors. The first approach advocates hardware support for shared memory. The second approach prefers to connect processing nodes, each containing commodity microprocessors and memory, with a fast network.

Due to the popularity of the shared memory model, many system designers include in their platforms hardware support for shared memory. At the low-end, shared memory is becoming ubiquitous in the form of small-scale symmetric multiprocessors (SMPs). In these platforms, the processors are attached on the same memory bus. Both commodity microprocessors and system logic are specifically designed to support shared memory using bus-based coherence protocols [SS86]. Consequently, it becomes easy to build low-end shared memory systems since the fixed design costs are amortized over the large sales volumes of commodity microprocessors. Therefore, these systems are achieving cost-performance superior to their uniprocessor counterparts.

Supporting shared memory beyond a small number of processors however, requires extensive custom hardware support due to inherent scalability bottlenecks of bus-based shared memory. Both physical limitations and the non-linear cost of memory bus bandwidth prevents SMPs from scaling directly to a large number of processors. High-end shared memory servers extend the bus-based coherence protocols across many SMP nodes using custom system logic. Most often, the system logic implements the distributed shared memory (DSM) paradigm that provides a shared-memory abstraction over the physically distributed memory in a parallel

machine. Many recent commercial systems such as Sequent STiNG [LC96] and SGI Origin [LL97] follow this approach. However, the associated fixed design costs lead to high premiums charged for these shared memory platforms.

A less costly way to realize the performance potential of parallel processing is to connect commodity microprocessors using a custom low-latency network. These parallel platforms have been called massively parallel processors (MPPs). Since relatively little extra hardware except the custom network is required, this approach has enjoyed popularity in older machines such as Intel iPSC860 [Int90] and TMC CM-5 [Thi91]. It is still present today in commercial systems such as IBM SP-2.

At a high level of detail, there is not much difference between MPPs and a collection of workstations with the exception of the custom network. Recently however, the network technology has caught up with the other components of a workstation, prompting researchers to advocate using networks of workstations (NOWs) as parallel "minicomputers" [ACP95]. NOWs can take advantage of commodity components to a greater extent than MPPs. Not only commodity microprocessors can be used, but all the system components including backplane busses, system logic, and peripherals can now be off the self equipment.

My thesis refers to networks of workstations as clusters of workstations (COWs). The difference between NOWs and COWs is more perceived than real. Berkeley, which coined the NOW term, envisioned NOWs as desktop workstations that can be used as a parallel platform when idle. In contrast, COWs are assumed to be workstations that are used as a dedicated platform for running sequential and parallel applications. In this way, job scheduling issues become less important. Moreover, you do not have to pay the overheads associated with implementing process migration policies that are necessary in the NOW environment. Nevertheless, such issues are orthogonal to the ones addressed in this thesis. Therefore, in my thesis, the two terms are considered interchangeable.

A key factor that enables the use of commodity network equipment in NOWs has been that the emergence of interfaces that provide protected user-level access to the network interface (NI), so the operating system need not be invoked in the common case (user-level messaging). Commercial designs such as Myricom Myrinet [BCF+95], DEC Memory Channel [GCP96], Tandem TNet [Hor95] have become widely available. Collectively, these designs have been called system-area networks [Bel96] to distinguish them from traditional local area networks that have higher messaging overheads. System-area networks also enable the construction of clustered shared memory servers targeted for high availability and scalability.

Due to the proliferation of parallel programming models and computing platforms, portability of parallel programs remains a significant concern for parallel programs. The emergence of standard message-passing libraries, such as PVM and MPI [GBD+94, For94], that have been implemented both for message-passing and shared memory platforms has addressed the portability issue for message-passing programs.

Shared memory has been limited to page-based systems [LH89,CBZ91]. *Shared virtual memory* (SVM) [LH89,CBZ91] is the most common form of software DSM and implements coherence at page granularity using standard address translation hardware found in commodity microprocessors. Such systems suffer from fragmentation and false sharing and can perform poorly in the presence of fine-grain sharing [EK89]. For acceptable performance, page-based systems often resort to weaker shared memory consistency models [KDCZ93,ENCH96,KHS+97,YKA96].

Shared virtual memory systems lack *fine-grain access control*, a key feature of hardware shared memory machines. Access control is the ability to selectively restrict reads and writes to memory regions. At each memory reference, the system must perform a *lookup* to determine whether the referenced data is in local memory, in an appropriate state. If local data does not satisfy the reference, the system must invoke a *protocol action* to bring the desired data to the local node. We refer to the combination of performing a lookup on a memory reference and conditionally invoking an action as *access control*. Access control *granularity* (also referred to as the *block size*) is the smallest amount of data that can be independently controlled. Access control is fine-grain if its granularity is similar to a hardware cache block (32-128 bytes).

FGDSM systems can implement fine-grain access semantics and the coherence protocol either in software or hardware. Hardware shared-memory machines achieve high performance by using hardware-intensive implementations, but this additional hardware is not available on message-passing platforms. Therefore, FGDSM systems on message-passing hardware must rely on techniques that require little or no additional hardware. Like SVM systems, these systems use address translation hardware to map shared addresses to local memory pages but enforce coherence at a finer granularity. FGDSM systems achieve performance competitive to SVM systems [ZIS+97] without having to resort to weak consistency models.

Blizzard/CM-5 [SFL+94], developed by the author and others for the TMC CM-5, was the first FGDSM system on messaging passing hardware. Blizzard/COW[1] [SFH+96], which is its direct descendant, was developed on the Wisconsin Cluster of Workstations (COW) and is the focus of this thesis. Digital's Shasta [SGT96] is another FGDSM system inspired by Blizzard/CM-5. Unlike Shasta, Blizzard uses the Tempest interface [Rei94] to support software distributed shared memory. Tempest separates the mechanisms required to implement distributed shared memory from the coherence protocols that enforce the shared memory semantics. By default, shared memory programs are linked against a library that implements an 128-byte S-COMA-like [HSL94] software protocol [RLW94] to maintain coherence across nodes.

Tempest's flexibility allows the implementation of application-specific custom coherence protocols [FLR+94] which integrate shared memory with messaging. Such protocols, called *hybrid* protocols, can reduce the overhead of fine-grain access control by being tailored to the

1. Throughout the thesis, Blizzard without qualification refers to the COW implementation. Blizzard/CM-5 is used to refer to the older Blizzard system.

application access patterns. In addition, Tempest provides an attractive target to build runtime systems for parallel languages [LRV94,CL96] where compilers can tailor the coherence protocol to incorporate high-level knowledge about the application access patterns. While other hardware shared memory systems integrate message passing and shared memory [HGDG94], none offers the same flexibility to develop application-specific protocols as user (rather than system) libraries.

High-end Tempest implementations such as the Typhoon designs [RLW94,RPW96] include extensive hardware support for fine-grain access control and protocol actions. Such designs achieve performance competitive to other hardware shared memory approaches [RPW96] but still offer Tempest's flexibility to support coherence protocols. Blizzard, however, implements coherence in software but maintains fine-grain access semantics either in software or hardware using mostly commodity hardware and software. Tempest's flexibility to support custom coherence protocols is especially important for Blizzard since the overhead of fine-grain access control, protocol actions, and messaging is higher with Blizzard than it is in hardware-intensive Tempest implementations. Blizzard also demonstrates the portability provided by the Tempest interface. Tempest allows clusters to support the same shared-memory abstraction as supercomputers, just as MPI and PVM support a common interface for coarse-grain message passing.

Table 1.1 summarizes the relationships between parallel programming models and parallel programming platforms as we have discussed them in this section. Modern parallel machines are designed using commodity microprocessors to directly support message-passing or shared memory. Hardware shared memory machines (SMPs, DSMs) can effectively—but not optimally—support message-passing or hybrid programming models through emulation. Experimental designs including Tempest high-end implementations better integrate shared memory and message-passing. Tempest implementations, however, are the only ones that support application-specific protocols as user-level libraries. Blizzard's domain is existing message-passing platforms such as MPPs, COWs, or clustered servers. In this domain, message-passing is directly supported by message-passing libraries. Shared memory, however, requires software with little or no additional hardware. Blizzard has the following two advantages over other approaches.

- It supports fine-grain shared memory programs as efficiently as SVM systems without having to resort to weaker consistency models.
- It supports the Tempest interface, which allows the development of application-specific hybrid coherence protocols, tailored to the application access patterns.

1.2 System Overview

This section presents background information on the Wisconsin COW platform, Blizzard's components, the Tempest interface, and Tempest's default shared memory coherence protocol (stache).

Table 1.1: Parallel programming platforms vs. parallel programming models.

Parallel Platform		Programming Model		
		Shared Memory	**Message Passing**	**Hybrids**
Shared memory hardware	**Existing SMPs &DSMs**	Direct support (designed)	Through shared memory	Through shared memory
	Experimental DSMs: Typhoon (Flash, Alewife)	Direct support (designed)	Direct support (designed)	Direct support (designed)
No shared memory hardware	**Message passing (PVM, MPI, AM)**	-	Direct support (designed)	-
	Page based DSMs	Poorly without weak consistency	-	-
	FGDSMs	Competitive to page-based DSMs without weak consistency	-	-
	Blizzard	Direct support (similar to FGDSMs)	Direct support (similar to message passing libraries)	Direct support

1.2.1 Wisconsin Cluster Of Workstations

The Wisconsin COW consists of 40 dual-processor Sun SPARCStation 20s. Each contains two 66 Mhz Ross HyperSPARC processors [ROS93] with a 256 KB L2 cache memory and 64

MB of memory. The cache-coherent 50 Mhz MBus connects the processors and memory. I/O devices are on the 25 Mhz SBus, which a bridge connects to the MBus. SPARCStation 20s provide an I/O virtual address space for Sbus devices with a separate MMU in the MBus-to-SBus bridge. The I/O MMU maps 32 bit Sbus addresses to 36 bit Mbus addresses using a one-level page table. Under Solaris 2.4, the I/O MMU directly maps kernel virtual addresses to Sbus virtual addresses, which limits DMA operations to the kernel address space. The operating system of the COW nodes is Solaris 2.4.

Each COW node contains a custom-built Vortex card [Pfi95] and a Myrinet network interface [BCF+95]. The Vortex card plugs into the MBus and performs fine-grain access control by snooping bus transactions. Each node also contains a Myrinet interface, which consists of a slow (7–8 MIPS) custom processor (LANai -2) and 128 KBytes of memory. The LANai performs limited protocol processing and schedules DMA transfers between the network and LANai memory or LANai memory and SPARC memory. The LANai processor cannot access SPARC memory directly using loads and stores. However, it can move data using DMA operations. Moreover, it can interrupt the host processor. The Myrinet switches are connected in a tree topology with the nodes attached as leaves of the tree.

Jobs intended to run on the COW processing nodes are submitted through the *Distributed Job Manager* (DJM) [Cen93]. The version of DJM running on the COW is based on the CM-5 1.0 version, but it has been heavily modified for the COW environment by Mark Dionne [Dio96].

1.2.2 Blizzard and FGDSM Resources

In Blizzard, shared memory is supported only for regions allocated through special *malloc()*-like calls that manage a shared heap[1]. Blizzard preallocates an address range within the application address space for the shared heap (Figure 1-1). Initially, accesses to the shared region are disabled by setting the page protection to *none* for all the pages in the region. On the first access to a page in the shared heap, a fault occurs which the operating system forwards to a Blizzard signal handler. Tempest specifies user-level page fault handlers and exports primitives which toggle the page protection to read-write and enable file-grain access control for that page.

Figure 1-2 illustrates the resources required to implement FGDSM systems. Shared data pages are distributed among the nodes, with every node serving as a designated *home* for a group of pages. A *directory* maintains sharing status for all the memory blocks on the home nodes. A *remote cache* serves as a temporary repository for data fetched from remote nodes. A set of *fine-grain tags* enforce access semantics for shared remote memory blocks. Upon access violation, the address of the faulting memory block and the type of access are inserted

1. Blizzard supports the PARMACS programming model [BBD+87]. PARMACS offers to each process of a parallel application a private address space with *fork*-like semantics. Shared memory support is limited to the special shared heap.

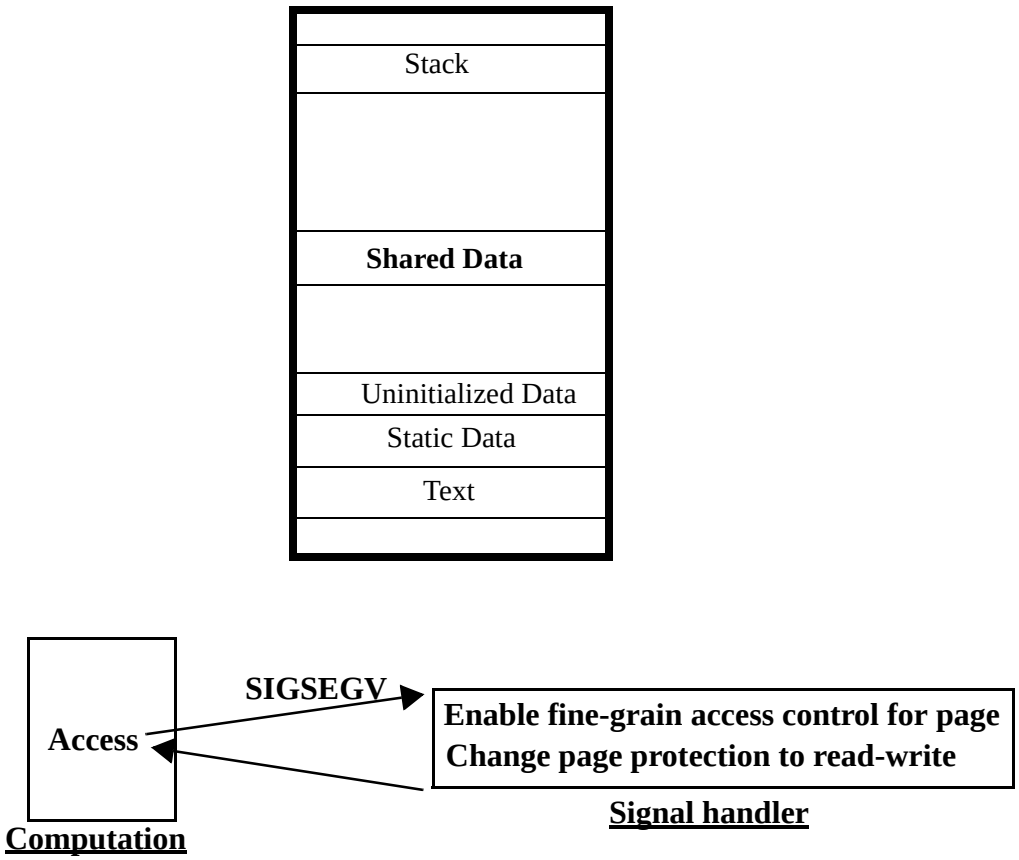

Figure 1-1. Process address space and initialization of shared pages.

in the *fault queue*. Processor(s) are responsible for running both the application and the software coherence protocol that implements shared memory. The protocol moves memory blocks among the nodes using *send message queues* and *receive message queues.*

Figure 1-2 also illustrates a breakdown of the resources accessed by the application and the protocol respectively. An application verifies access semantics upon a memory operation by reading the fine-grain tags. Memory operations also may read or write data to either home pages or the remote cache. An access violation in the application will insert an entry into the fault queue. The protocol manipulates the fine-grain tags to enforce access semantics to shared data, manages the remote cache data and state, maintains the list and status of sharers in the directory, removes entries from the fault queue, and sends and receives messages.

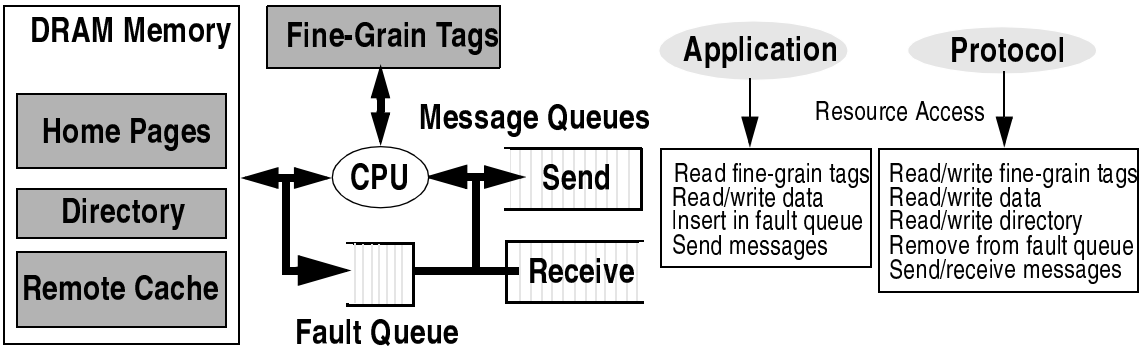

Figure 1-2. Resources in a software FGDSM system.

1.2.3 Tempest

Tempest is an interface that enables user-level software to control a machine's memory management and communication facilities [Rei94]. Software can compose the primitive mechanisms to support either message-passing, shared-memory, or hybrid applications. Since these primitives are available to programs running at user level, a programmer or compiler can tailor memory semantics to a particular program or data structure, much as RISC processors enable compilers to tailor instruction sequences to a function call or data reference [Wul81]. Tempest provides four types of mechanisms:

Virtual Memory Management. With user-level virtual-memory management, a compiler or run-time system can manage a program's address space. Tempest provides memory mapping calls that the coherence protocol uses to manage a conventional flat address space. In the context of a commodity operating system, this mechanism is trivially implemented using system calls that change the page protection for the shared heap.

Fine-Grain Access Control. Fine-grain access control is the Tempest mechanism that detects reads or writes to invalid blocks or writes to read-only blocks and traps to user-level code, which uses the resulting exceptions to execute a coherence protocol action [HLW95]. At each memory reference, the system must ensure that the referenced datum is local and accessible. Therefore, each memory reference is logically preceded by a check that has the following semantics:

```
if (!lookup(Address))
   CallHandler(Address, AccessType)
memory-reference(Address)
```

If the access is allowed, the reference proceeds normally. Otherwise, the shared-memory protocol software must be invoked. This sequence must execute atomically to ensure correct-

ness. In the Tempest model, fine-grain access control is based on tagged memory blocks. Every memory block—an aligned, implementation-dependent, power-of-two size region of memory—has an access tag of *ReadWrite*, *ReadOnly*, or *Invalid* that limits the allowed accesses. Reads or writes to a *ReadWrite* block or reads to a *ReadOnly* block complete normally, but an access to an *Invalid* block or a write to a *ReadOnly* block causes a block access fault. These faults are similar to page faults. They suspend the thread making the access and invoke a user-level handler. A typical handler performs the actions dictated by a coherence protocol to make the access possible and updates the tag. Then the access is retried.

Low-Overhead Messaging. Low-overhead "active" messages [vECGS92] provide low-latency communication, which is fundamental to the performance of many parallel programs. In Tempest, a processor sends a message by specifying the destination node, handler address, and a string of arguments. The message's arrival at its destination creates a thread that runs the handler, which extracts the remainder of the message with *receive* calls. A handler executes atomically with respect to other handlers, to reduce synchronization overhead.

Bulk Data Transfers. Efficient transfers of large quantities of data is critical for many applications. In Tempest, a processor initiates a bulk data transfer much as it would start a conventional DMA transaction, by specifying virtual addresses on both source and destination nodes. The transfer is logically asynchronous with the computation.

1.2.4 Stache

Tempest's default coherence protocol is an 128-byte S-COMA-like [HSL94] software protocol [RLW94] called stache. Stache maintains internal data structures to correctly implement a sequentially consistent shared memory coherence protocol.

For each shared page, one node acts as the home node. By default, home pages are distributed among the nodes in a round-robin manner, but sophisticated programs are able to optimize the allocation using a first-touch policy. With this policy, all home pages are allocated on the first node during the initialization phase. After the parallel phase begins, any access to a page by a node, will result in that node becoming the home node for the page. This allocation policy results in improved performance because normally stache is a four-hop protocol unless the home node is one of the sharer nodes. More specifically, the first time that some node other then the home node accesses a memory location in a page, it will fault. The stache fault handler will request a copy of the block from the home node. If the block is not available at the home node, it will be forwarded to the node that has the most recent copy. That node will send the block back to the home. From there, it will be forwarded to the requesting node. Since there four messages are required to service this request, stache is a four-hop protocol. If no other nodes were involved only two messages need to be exchanged.

1.3 Thesis Overview

The thesis examines the issues involved in supporting fine-grain shared memory in clusters of workstations. The thesis organization is as follows. Chapter 2 discusses fine-grain access control implementations on commodity workstations. Chapter 3 discusses issues in the design of messaging subsystems for FGDSM systems. Chapter 4 presents performance results and evaluates Blizzard as a platform for parallel applications. Chapter 5 discusses the advantages of multiprocessor workstations for FGDSM systems. Chapter 6 discusses address translation in network interfaces to support zero-copy messaging directly to user data structures. Chapter 7 presents a summary of the thesis, discusses the implications of this work and points to possible future directions. A brief summary of the main points in each chapter of the thesis follows.

Chapter 2 investigates fine-grain access control implementations on commodity workstations. It presents four different fine-grain tag implementations with different performance characteristics. The first, Blizzard-S, adds a fast lookup before each shared-memory reference [LW94] by modifying the program's executable [LB94]. The second, Blizzard-E, uses the memory's error-correcting code (ECC) bits as valid bits and the page protection to emulate read-only bits [RFW93]. The third, Blizzard-ES, combines the two techniques using software lookups for stores and ECC for loads [SFL+94]. The fourth, Blizzard/T, uses Vortex, a custom fine-grain access control accelerator board. The board was originally developed by members of the Wisconsin WindTunnel project [Pfi95] to demonstrate the simplest possible hardware Tempest implementation (Typhoon-0 [RPW96]). Blizzard/T explores the Vortex's ability to support hardware fine-grain tags modified without kernel intervention but unlike Typhoon-0, it does not use any of its other abilities to accelerate Tempest operations (e.g., mechanisms to accelerate dispatch of access fault handlers).

Chapter 2 makes the following three contributions.

- First, it identifies fine-grain access control as a memory property that can be associated either with the physical or the virtual memory. It argues that the fine-grain access control tags should be associated with the physical address space for performance, but physical addresses should not be exposed to the user so that the kernel is not restricted in changing the binding of virtual to physical addresses.
- Second, it describes the kernel support that the hardware techniques require to access hardware resources beyond those the operating system was designed to support. Blizzard's kernel support offers extended kernel functionality through runtime-loadable modules, follows a modular approach, and optimizes all the performance-critical operations.
- Third, it presents low-level performance results to evaluate the different Blizzard fine-grain access control implementations.

Chapter 3 investigates messaging subsystem design for FGDSM systems. For standard shared memory coherence protocols, where data are transferred in cache-block sized quantities, low-latency communication is critical to the performance of many parallel programs. For application-specific protocols, where the coherence protocol has been optimized to fit the

application communication requirements, high bandwidth is of equal concern to low latency. Tempest's messaging interface is based on the Berkeley active messages [vECGS92], but it differs in two important aspects, necessary to use Tempest messages to implement *transparent* shared memory protocols. Tempest messages are not constrained to follow a request-reply protocol and are delivered without explicit polling by an application program.

Chapter 3 makes the following four contributions.

- First, it develops a methodology to compute the buffer requirements for an arbitrary messaging protocol. Using this methodology, it shows that shared memory coherence protocols, while not pure request/reply protocols, have bounded buffer requirements.
- Second, it evaluates buffer allocation policies for active message implementations. It proposes a buffer allocation policy that does not require buffers in a node's local memory in the common case, yet is robust enough to handle arbitrary traffic streams.
- Third, it evaluates different solutions to avoid explicit polling and it proposes implicit polling using binary rewriting to insert polls in the application code.
- Fourth, it present performance results that indicate that the extended functionality does not affect performance adversely.

Chapter 4 presents performance results to evaluate Blizzard as a platform for running parallel applications. For this task, both low-level microbenchmarks and application benchmarks are used. The performance results suggest that parallel applications with small communication requirements easily achieve significant speedups using transparent shared memory. Parallel applications with high communication requirements however, require application-specific coherence protocols, selectively employed on the performance-critical data structures. Moreover, the results show that application-specific protocols are particularly appropriate for FGDSM systems on low-cost platforms because their relative performance benefit is significantly higher than other proposed tightly-integrated high-end Tempest implementations.

Chapter 5 investigates extending FGDSM systems to networks of multiprocessor workstations. FGDSMs are particularly attractive for implementing distributed shared memory on clusters of multiprocessor workstations because they transparently—i.e., without the involvement of the application programmer—extend the fine-grain hardware shared-memory abstraction across a cluster. Grouping processors into multiprocessor nodes allows them to communicate within the node using fast hardware shared-memory mechanisms. Multiple processors can also improve performance by overlapping application execution with protocol actions. However, simultaneous sharing of node's resources (e.g., memory) between the application and the protocol requires mechanisms for guaranteeing atomic accesses [SGA97]. Without efficient support for atomicity, accesses to frequently shared resources may incur high overheads and result in lower performance with multiprocessor nodes.

Chapter 5 makes the following two contributions.

- First, it identifies the shared resources in FGDSM systems to which access should be controlled in an multiprocessor environment and it proposes techniques to address the syn-

chronization issues for each resource based on the frequency and type of accesses to that resource.

- Second, it compares the performance of Blizzard's uniprocessor-node and multiprocessor-node implementations while keeping the aggregate number of processors and amount of memory constant. The performance results suggest that grouping processors: (i) results in competitive performance while substantially reducing hardware requirements, (ii) benefits from custom hardware support for fine-grain sharing, (iii) boosts performance in FGDSMs with high-overhead protocol operations, and (iv) can hurt performance in pathological cases due to the ad hoc scheduling of protocol operations on a multiprocessor node.

Chapter 6 investigates address translation mechanisms in network interfaces to support zero-copy messaging directly to user data structures and avoid redundant data copying. Good messaging performance is important for application-specific protocols that push the limits of the messaging hardware. For zero-copy messaging, the network interface must examine the message contents, determine the data location and perform the transfer. The application accesses data using virtual addresses, which can be passed to the interface when the message operation is initiated. However, the network interface accesses memory using physical addresses, so the virtual address known by the application must be translated to a physical address usable by the network interface.

Chapter 6 makes the following four contributions.

- First, it presents a classification of address translation mechanisms for network interfaces, based on where the lookup and the miss handling are performed.
- Second, it analyzes the address translation design space and it evaluates a series of designs with increasingly higher operating system support requirements. For each design point, it determines whether it can accelerate messaging and whether it degrades gracefully once the translation structures thrash, considers the required operating support and proposes techniques for graceful degradation in the absence of appropriate operating system interfaces.
- Third, it provides performance data from simulations that demonstrate that even without operating system support, there exist solutions so that the performance properties hold.
- Fourth, it demonstrates the feasibility of the approach by presenting experimental results from an implementation on real hardware within the Blizzard framework.

Chapter 2

Fine Grain Access Control

Fine-grain access control is a fundamental operation in shared memory systems [SFL+94]. Regardless of how shared memory is implemented, memory accesses are checked against the current coherence protocol state and protocol actions are initiated to deal with accesses that are not allowed in terms of the protocol state. Blizzard provides shared memory and consequently, must implement these two operations. The design and implementation of Blizzard's fine-grain access control mechanisms is the focus of this chapter.

Blizzard supports the Tempest interface [Rei94], which exposes fine-grain access control tags to user level, allowing shared memory coherence protocols to be implemented as user-level libraries. Tempest's fine-grain access control mechanism detects reads or writes to invalid blocks or writes to read-only blocks and traps to user-level code, which uses the resulting exceptions to execute a coherence protocol action in software [HLW95]. At each memory reference, the system must ensure that the referenced datum is local and accessible. If the access is allowed, the reference proceeds normally. Otherwise, the shared-memory protocol software must be invoked, which is achieved by suspending the computation and invoking a user-level handler. A typical handler performs the actions dictated by a coherence protocol to allow the access and then resumes the computation.

The fine-grain access control mechanism is similar to full/empty bits of dataflow architectures [DCF+89] but it is tailored to support the implementation of shared memory protocols. For this reason, it extends the two-state model of the full/empty bits to a three-state model that includes a readonly state. More specifically, Tempest's fine-grain access control is based on tagged memory blocks. Every memory block (an aligned, implementation-dependent, power-of-two size region of memory) has an access tag of *ReadWrite*, *ReadOnly*, or *Invalid*

that limits the allowed accesses. For example, reads or writes to a *ReadWrite* block or reads to a *ReadOnly* block complete normally. However, an access to an *Invalid* block or a write to a *ReadOnly* block causes an access fault.

The performance of a shared memory implementation depends both on the overheads of the access check and the protocol action operations. Depending on how these operations are implemented, they will have different performance, cost and design characteristics. We can classify the implementation techniques in software, hardware or combination approaches. Hardware techniques provide better performance than software techniques but at a higher cost point. Since the access check and the protocol action allow different implementations, system designers can follow alternative paths in the design space. Blizzard represents a low cost design point that relies mainly on software to implement these operations. This chapter reviews the design space and places Blizzard into perspective to other approaches.

Since Tempest exposes fine-grain access control tags to the user level, the fine-grain access control information becomes user-visible and modifiable state. This state is a attribute that accompanies the application memory and therefore, it must be integrated within other abstractions related to memory. More specifically, general purpose operating systems virtualize the physical memory: user processes do not directly refer to physical memory but instead, they use addresses that refer to a virtual address space. Virtual memory references from applications are translated to physical memory references by the address translation hardware of modern processors. Consequently, memory properties are associated either with the physical or the virtual memory. The question arises with which one should we associate fine-grain access control. I argue that the fine-grain tags should be associated with the physical address space for performance, but physical addresses should not be exposed to the user and used to refer to the tags so that the kernel is not restricted in changing the binding of virtual to physical addresses.

Blizzard implements fine-grain access control using a variety of techniques. Common to all techniques is that the protocol actions are executed on the node processors in software. What differentiates these techniques is the way with which the fine-grain access control tags are implemented. We can distinguish software and hardware techniques. The software techniques rely on explicit checks inserted in the instruction stream to enforce the appropriate semantics. The hardware techniques use commodity or custom hardware to accomplish the same task while avoiding the overhead of explicit access checks.

Hardware techniques require kernel support to access hardware resources beyond those the operating system was designed to support. Blizzard runs under a Solaris 2.4 kernel, which it extends to support hardware fine-grain access control. The following three design principles characterize this effort. First, a constant constraint is the use of an unmodified commodity kernel. Extended functionality is offered through runtime-loadable kernel object modules. Second, a modular approach is followed. A hardware-independent module extends internal kernel interfaces while hardware-specific device drivers rely on its services to support specific hardware techniques. Third, all the performance-critical operations are extensively optimized.

The rest of this chapter is organized as follows. Section 2.1 discusses fine-grain access control as a fundamental operation in shared memory systems. Moreover, it defines a design space for its implementation, in which Blizzard is placed. Section 2.2 portrays fine-grain access control as a memory attribute and discusses its integration within the existing memory abstractions. Section 2.3 focuses on the software and hardware techniques employed by Blizzard to provide fine-grain access control in the Wisconsin Cluster Of Workstations (COW). Section 2.4 presents the kernel support for hardware fine-grain access control, which has been implemented in Sun's Solaris 2.4 operating system. Section 2.5 presents low level performance results to evaluate the different fine-grain tag implementations. Section 2.6 finishes the chapter with the conclusions of this study.

2.1 Fine Grain Access Control Design Space

Fine-grain access control can be implemented in many ways. The lookup of the current state and/or the protocol action can be performed in either software, hardware, or a combination of the two. These alternatives have different performance, cost, and design characteristics. This section classifies access control techniques based on where the lookup is performed and where the action is executed, and places Blizzard within the design space.

Either software or hardware can perform the access check. A software lookup avoids the expense and design cost of hardware, but incurs a fixed overhead on each lookup. Hardware typically incurs no overhead when the lookup does not invoke an action. Lookup hardware can be placed at almost any level of the memory hierarchy: TLB, cache controller, memory controller, or a separate snooping controller.

When a lookup detects a conflict, it must invoke an action dictated by a coherence protocol to obtain an accessible copy of a block. As with the lookup itself, hardware, software, or a combination of the two can perform this action. With software, the protocol action can execute either on the same processors as the application or on a dedicated processor.

Tempest differs from other approaches to shared memory because it exposes fine-grain access control to user level. This allows the implementation of coherence protocols as user-level libraries written in a general-purpose programming language. The extra flexibility however, restricts possible implementations to designs where the protocol actions are executed in an environment sufficiently similar to the one that the computation is executing (i.e., a processor).

Reinhardt, et al., have shown that the overheads to support user-level coherence protocols do not inhibit high-end Tempest implementations, such as the proposed Typhoon designs, from being competitive to dedicated hardware solutions [RPW96,Rei96]. Such implementations, targeted for the server market, emphasize performance over cost. These systems provide hardware support for both the access control test and protocol action. For example, a dedicated processor for protocol actions can minimize invocation and action overhead while still exploiting commodity computation processors. However, this approach requires either a com-

plex ASIC or full-custom chip design, which significantly increases design time and manufacturing cost.

Blizzard and Digital's Shasta [SGT96], on the other hand, are FGDSM systems targeted toward networks of workstations or personal computers. In such environments, the cost and complexity of additional hardware is more important because they must compete on uniprocessor cost/performance. Blizzard relies mainly on commodity software and hardware to provide fine-grain access control. However, simple custom hardware support for the access test can be cost-effective and therefore, it is also considered. Previous approaches to shared memory on networks of workstations mainly targeted page-based distributed shared memory (DSM) systems [LH89]. Such systems required relaxed memory consistency models, which complicate the programming model, to overcome the effects of false sharing and fragmentation and offer acceptable performance [KDCZ93]. Zhou, et al., however, have shown that sequentially consistent fine-grain DSM systems are competitive with page-based systems that support weaker consistency models [ZIS+97].

Blizzard explores four different fine-grain tag implementations. These designs cover a substantial range of the design space for fine-grain access control mechanisms [SFL+94]. Blizzard/S uses a software method based on locally available executable-editing technology [LS95]. Blizzard/E uses commodity hardware to accelerate the access check. The memory controller supports *Invalid* blocks while the TLB is used to emulate *ReadOnly* blocks. Blizzard/ES also uses the memory controller for *Invalid* blocks but relies on the same software method as Blizzard/S for *ReadOnly* blocks. Finally, Blizzard/T uses Vortex, a custom board [Pfi95] to accelerate many fine-grain access control operations.

We shall subsequently review alternative techniques to implement the access check (Section 2.1.1) and alternative locations to execute the protocol actions (Section 2.1.2).

2.1.1 Access Check

Software. The code in a software lookup checks a main-memory data structure to determine the state of a block before a reference. Static analysis can detect and potentially eliminate redundant tests or even batch accesses to the consecutive cache blocks as in Shasta [SGT96]. Either a compiler or a program executable editing tool [LB94,LS95] can insert software tests. With the latter approach every compiler need not reimplement test analysis and code generation. Blizzard/S and Shasta [SGT96] also follow this approach (Section 2.3.1). Compiler-inserted lookups can exploit application-level information. Orca [BTK90], for example, provides access control on program objects instead of blocks.

Translation Lookahead Buffer (TLB). Standard address translation hardware provides access control, though at memory page granularity. Nevertheless, it forms the basis of several distributed-shared-memory systems such as IVY [LH89], Munin [CBZ91] and TreadMarks [KDCZ93]. Blizzard/E partially relies on the TLB to emulate fine-grain access control (Section 2.3.2). Though unimplemented by current commodity processors, additional,

per-block access bits in a TLB entry could provide fine-grain access control. The “lock bits” in certain IBM RISC machines, including the 801 [CM88] and RS/6000 [OG90], provide access control on 128-byte blocks. Unfortunately, these bits can only partially support fine-grain access control since they do not provide the *ReadOnly* state. Tamir, et al., have proposed extensions to the format of the page table in order to fully support fine-grain access control [TJ92].

Cache Controller. The MIT Alewife [CKA91] and Kendall Square Research KSR-1 [Ken92] shared-memory systems use custom cache controllers to implement access control. In addition to detecting misses in hardware caches, these controllers determine when to invoke a protocol action. On Alewife, a local directory is consulted on misses to local physical addresses to determine if a protocol action is required. Misses to remote physical addresses always invoke an action. Due to the KSR-1’s COMA architecture, any reference that misses in the remote memory cache requires protocol action. A trend toward on-chip second-level cache controllers [Hsu94] has made modified cache controllers incompatible with modern commodity processors such as the Pentium Pro [Int96] and PowerPC 750 processors.

Memory Controller. If the system can guarantee that the processor’s hardware caches never contain *Invalid* blocks and that *ReadOnly* blocks are cached in a read-only state, the memory controller can perform the lookup on hardware cache misses. This approach is used by Sun’s S3.mp [NMP+93], NIMBUS’s NIM 6133 [NIM93,SFL+94], and Stanford’s FLASH [K+94]. S3.mp has a custom memory controller that performs a hardware lookup at every bus request. NIM 6133 integrates fine-grain access control with memory’s error correction code. FLASH’s programmable processor in the memory controller performs the lookup in software. It keeps state information in regular memory and caches it on the controller. Custom controllers are not possible, however, with many current processors. Blizzard/E and Blizzard/ES partially rely on existing support for error correction in commodity microprocessors. Microprocessor trends make it increasingly difficult to use this mechanism with many future microprocessors that cannot recover from late arriving exceptions in the lower levels of the memory hierarchy.

Bus Snooping. For economic and performance reasons, most hardware approaches avoid changes to commodity microprocessors. Nevertheless, most modern commodity processors support bus-based coherence protocols and therefore, implement fine-grain access control in the processor cache controller. Despite the progress in bus-based symmetric multiprocessors, evident in designs such as Sun’s Enterprise Server 6000, the scalability of bus-based systems is ultimately limited. When a processor supports a bus-based coherence scheme, a separate bus-snooping agent can perform a lookup similar to that performed by a memory controller. Stanford DASH [LLG+92] and Typhoon [RLW94] among experimental designs, employ this approach. Many recent commercial shared memory machines, such as Sequent STiNG [LC96] and SGI Origin [LL97] also follow this approach.

2.1.2 Protocol Action

Custom Hardware. High performance shared memory systems use dedicated hardware to execute the protocol actions. The list includes older systems like DASH, KSR-1, and S3.mp as well as newer systems like Convex Exemplar and SGI Origin. Dedicated hardware provides high performance for a single protocol. While custom hardware performs an action quickly, research has shown that no single protocol is optimal for all applications [KMRS88] or even for all data structures within an application [BCZ90,FLR+94]. Hybrid hardware/software protocols such as Alewife's LimitLESS [CKA91], Dir_1SW [HLRW93], and Typhoon-2 [Rei96] implement the expected common cases in hardware and trap to system software to handle complex, infrequent events. High design costs and resource constraints make custom hardware unattractive. Nevertheless, as the relatively large number of commercially available systems suggests, system vendors believe that the costs are justified by the premiums charged for high-end systems.

Primary processor. Performing actions on the node processor(s) provides protocol flexibility and avoids the additional cost of custom hardware or a dedicated processor. Blizzard uses this approach as do page-based DSM systems such as IVY and Munin. However, interrupting an application to run an action can add considerable overhead. Alewife addressed this problem with a modified processor that supports rapid context switches. Blizzard implementations include extensive operating system support to minimize the context switch overhead.

Dedicated processor. FLASH, Sequent STiNG, and Typhoon achieve both high performance and protocol flexibility by executing actions on an auxiliary processor dedicated to that purpose. This approach avoids a context switch on the node processor(s) and may be crucial if the processor(s) cannot recover from late arriving exceptions in the lower levels of the memory hierarchy. In addition, an auxiliary processor can provide rapid invocation of action code, tight coupling with the network interface, special registers, and special operations. Of course, the design effort increases as the processor is more extensively customized. Blizzard can also dedicate a processor for execution of protocol actions. However, studies have shown that for networks of workstations this policy generally does not result in the best utilization of the processor [FW96,FW97,Fal97].

2.2 Fine-Grain Access Control Abstraction

This section examines the integration of the fine-grain access control as a memory attribute within other memory-related abstractions. When the operating system virtualizes a hardware resource, it exposes an abstraction of the resource that the user processes can access. This allows the manipulation of the resource and the enforcement of policies without the user processes being directly aware of the virtualization. In general-purpose operating systems, memory is one of the most important resources that are virtualized. User processes do not directly refer to physical memory but instead, they use addresses that refer to a virtual address space. Virtual memory references are translated to physical memory references by the address trans-

lation hardware of modern processors. Consequently, memory attributes can be associated either with the physical or the virtual memory.

In defining the association for each memory attribute, there are two key questions that we need to answer. First, we must decide how the user refers to the attribute. I shall call the handle that it is exposed to the user and it is used to to refer to the attribute (e.g., in library calls) as the *attribute name*. The attribute name can be a physical address or a virtual address. Second, we must decide whether the attribute value is associated with the physical or the virtual address space. I shall call the calue of the attribute as the the *attribute content*. Again, the attribute name can be associated with the virtual or the physical address space.

Table 2.1: Classification of memory attributes according to name and content.

Name vs. Content	Virtual Content	Physical Content
Virtual Name	Page Protection, Copy-On-Write Memory Pages	SysV Shared Memory Pages
Physical Name	-	Physical Memory Pages

Table 2.1 lists typical memory attributes and their classification according to this scheme. For example, the page protection is a memory attribute for which both the name and the content are associated with the virtual address space. When the content is associated with the virtual address space, it accompanies the virtual address and the attribute does not have any meaning independent of the mapping. Another example of an attribute with the same behavior are the pages in a memory region mapped with copy-on-write semantics. While the kernel can optimize memory usage by keeping the same physical page associated with different virtual addresses, as soon as an operation is performed that would reveal this fact, a new copy the page is created. The SysV shared memory interface [Vah96] exposes memory pages for which the name is associated with the virtual address space but the content is associated with the physical address space. In this case, a modification through one virtual name will be visible through other virtual names. Therefore, the attribute content is associated with the physical address space. Physical memory regions have both a physical name and a physical content. The kernel will use the physical name itself and pass it to device drivers or between kernel modules. In general however, OS's are reluctant to reveal physical names to the user processes. If the user process knows and can use the physical name of a resource, then the OS is severely restricted to the degree that it can virtualize this resource. For example, in standard operating systems there is not an interface to expose to the user the physical address of a memory page. If the physical address is known, the operating system cannot move the page without informing the user processes. This introduces additional complexity and overhead in the kernel.

Given the discussion so far, we are ready to examine fine-grain access control. An effective virtualization requires the name to be associated with the virtual address space. This decision precludes exposing names in the physical address space, as is being done in the Typhoon prototypes [Rei96]. However, it also precludes hardware acceleration of access faults using snooping devices if the virtual address is not readily available on the bus.

As for the attribute content, an example helps illustrate the implications of alternative choices. Suppose we have two virtual memory mappings for the same physical memory. Can we modify the fine grain access control information for the two mappings independently of each other? If it is associated with the virtual address space, the answer is yes. If it is associated with the physical address space, the answer is no. It can be argued that as with page protection, fine-grain access control should be associated with the virtual address space. However, there are two arguments against such an approach.

The first argument is circumstantial. No case has been identified where such an attribute is required; certainly not within the context of fine grain distributed shared memory systems. With hardware fine-grain tag implementations, a second memory mapping is used to access the same physical memory that it does not have any fine grain control state associated with it. Accesses through such mappings are not checked for validity. It can be argued that this is a limited form of virtualizing the content of the fine grain access control attribute. However, this mapping has very restricted semantics and it is only used internally by Blizzard without being exposed to the user. The second argument is pragmatic. The page protection is virtualizable because there is extensive hardware support in the memory management unit (MMU) of modern microprocessors. Even for the private memory with copy-on-write semantics, the same MMU hardware that is used to make the optimization possible. Without hardware support, it would be very expensive to virtualize the fine grain control information. Of course, pure software techniques such as the ones used by Blizzard/S can associate fine-grain access control with the virtual address space. An environment where one would consider virtualizing the attribute's content is with IBM RISC processors that supports "lock bits" in the TLB. In that case, hardware support makes it realistic to consider this alternative. Blizzard has been designed to be portable across a range of fine-grain tag implementations. Therefore, it is preferable to treat fine-grain access control as a memory attribute with a virtual name and a physical content.

The approach followed in Blizzard deviates from the Tempest specification [Rei94]. Tempest, like Blizzard, associates the attribute name with the virtual address space and the attribute content with the physical address space. However, Tempest distinguishes among the memory mappings to the same physical page. One is considered the primary mapping. An access faults through any virtual mapping is reported as occurring for the primary mapping. In other words, this approach ties the name of the fine grain access control information to the physical page, represented by the primary mapping. For pages shared across address space, the ramification is that one process would have to receive all the faults. Such behavior is more appropriate for debuggers[1] or monitor tools that need to take control when a fault is encountered by the process that they monitor. Its integration with other memory properties is awk-

ward. In addition, its implementation is problematic since access faults cease being synchronous faults and instead, require an asynchronous interrupt mechanism to notify the process that owns the primary mapping.

2.3 Blizzard Fine Grain Access Control Implementations

This sections presents the four Blizzard fine-grain tag implementations in detail. Each implementation represents a different point in the design space for fine-grain access control implementations and introduces different oveheads in the check overhead, handler invocation and access control manipulation.

2.3.1 Blizzard/S: Binary Rewriting

In this technique, access control tests are inserted before shared-memory loads and stores in a fully compiled and linked program. These tests use an instruction's effective address to index a table. The table contains the access control information (a byte per cache block) that determines if the memory reference should invoke a handler.

This technique descends from the Fast-Cache technique used in the simulation of memory systems [LW94]. It was first introduced in Blizzard/CM-5. In that system, it required 15 instructions (18 cycles on the CM-5 SPARC processors) before every load and store instruction [SFL+94]. The tool for inserting the checks originated from the *vt* tool, which in turn originated from *qpt*, a general purpose profiling tool [LB94]. The *vt* tool was used in the Wisconsin WindTunnel (WWT) simulator [RHL+93] to insert instructions that tracked the simulated time in target executables. It was not a sophisticated tool since it could only process the executable on a basic block basis. Therefore, it was not capable of any optimizations that depend on the analysis of the program's flow control graph. For example, it could not determine whether the condition codes were alive or not at any particular moment. Therefore, the inserted code had to avoid changing the Sparc condition codes or it had to resort in an elaborate (and slow) sequence of instructions to save them. For this reason, access checks were implemented using an indirect jump that its target was calculated based on the fine-grain tag value. In this way, condition codes were not modified. A similar access check sequence is still used today when better alternatives cannot be applied (Figure 2-1 (c)).

To address these problems, Eric Schnarr developed a new binary rewriting tool [SFH+96] based on the EEL executable rewriting library [LS95]. Besides software access control, I have been using this tool for rewriting Blizzard executables for other purposes as well (e.g., implicit network polling; see Chapter 3). The EEL library provides extensive infrastructure to analyze and modify executables. It builds the flow control graph of the executable and provides operators to traverse it and modify it. Furthermore, the library provides certain amenities that simplify the task of developing instruction sequences. It can determine whether

1. In retrospect, this is not surprising given the influence of the Wisconsin Wind Tunnel's *supervisor interface* [RFW93] on the design of Tempest.

register values are alive at any particular instance in the program, it can reassign registers so that the inserted code uses free registers and it can spill registers automatically when free registers are not available.

Using the control flow graph, the new tool introduces two important optimizations. First, it can eliminate memory accesses that cannot ever access shared memory (stack and global variables). Second, it can determine whether the condition codes are alive. This allows the use of faster access checks that modify the condition codes (Figure 2-1 (a)). The access check overhead is reduced to five instructions (six cycles) when the access is allowed (common case). Moreover, the use of the condition codes enables another important optimization, *sentinel values* [SFH+96]. When a block is invalid, its contents are overwritten with a specific value (*sentinel*) that acts as an invalid flag. Loads to shared memory proceed without checking in advance. The returned value, however, is checked against the sentinel (Figure 2-1 (b)). The fine-grain tags are consulted only in the case of a match. This check verifies the sentinel value and allows the technique to work correctly, even in the unlikely case that the actual value stored in memory was equal to the sentinel. The sentinel optimization reduces the check overhead to three instructions (three cycles) in the likely case of no match and it applies to most loads.

The access fault overhead for Blizzard/S is as low as a couple of function calls. When a fault is detected, a handler is invoked in the Blizzard runtime system. In Blizzard systems targeted for networks of workstations with uniprocessor nodes, this handler directly invokes the user protocol handler. The event sequence is slightly more complicated scheme with multiprocessor nodes (Chapter 5).

In Chapter 1, I described the general structure of the application address space for shared memory programs with Blizzard. Figure 2-2 elaborates on the structure of the application address space for Blizzard/S. On initialization, Blizzard/S uses the Unix system call *mmap()* to map the pseudo-device *"/dev/zero"* to the shared heap with the page protection set to *none*. This device simply supplies zero-filled pages. Due to the size of the heap (one GByte), the flags to *mmap()* request that no swap space should be reserved in advance. The fine-grain tag table is stored just next to the shared memory heap and it allocated also using *"/dev/zero"* with the protection set to *read-write*. Since the zero tag value corresponds to the *ReadWrite* state, on the first access to a shared heap page, the access check will succeed but a segmentation violation is caught by the kernel and the appropriate signal handler is invoked. The signal handler calls the user-supplied protocol page fault handler which can initialize the fine-grain tags and set the page protection to read-write for subsequent accesses.

2.3.2 Blizzard/E: Memory Error Correction Code & Page Protection

Blizzard/E technique uses deliberately incorrect error-correcting code (ECC) bits to detect invalid references to memory. It is representative of fine-grain access control mechanisms in which access checks happen in hardware but it may require system calls to manipulate the access control state. The error correction code has been previously used for fine-grain access

```
add addr, %g0, %g5                ! Get the effective address
srl %g5, log2(block_size), %l0    ! Calculate block byte index
ldub [%g6 + %l0], %l1             ! Load block state byte
andcc   %l1, 0x80, %g0            ! Test fine-grain bits
                                  ! 0xC0 is used for stores
be,a    next                      ! Avoid jump to handler
sub %l1, table base, %l0          ! Calculate handler table index
jmpl %g6 + %l0, %g6               ! Jump to handler jump table
nop
next:
```

(a) Fast sequence for loads and stores (uses condition codes)

```
subcc %g0, sentinel, %g5          ! Compare to sentinel
bne,a   next                      ! Avoid jump to handler
nop
jmpl %g6 + %l0, %g6               ! Jump to handler jump table
nop
next:
```

(b) Sentinel sequence for loads (uses condition codes)

```
add addr, %g0, %g5                ! Get the effective address
srl %g5, log2(block_size), %l0    ! Calculate block byte index
ldub [%g6 + %l0], %l1             ! Load block state byte
sub %l1, table base, %l0          ! Calculate handler table index
jmpl %g6 + %l0, %g6               ! Jump to handler jump table
nop
```

(c) Slow sequence for loads and stores (does not use condition codes)

Figure 2-1. Software Access Control Instruction Sequences.

The figure depicts the instruction sequences inserted in the application code before memory accesses using executable editing to implement software fine-grain access control. The sentinel sequence is inserted after the memory operations while the other two are inserted before the memory operation.

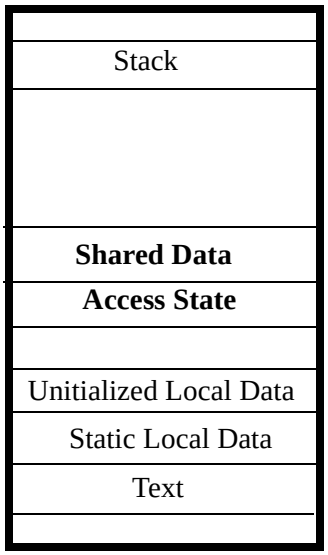

Figure 2-2. Process address space with Blizzard/S.

The figure displays the segments that comprise the address space of Blizzard processes under Blizzard/S. To support shared memory, two new segments are added to the traditional segments of a Unix process. The "shared data" segment is the address region for which shared memory is supported. The "access state" segment contains the fine-grain tags.

control in Blizzard/CM-5 [SFL+94] and in the simulation of memory systems [RHL+93, UNMS94]. The challenge in this case was to incorporate it in a commodity multiprocessor operating system. Since Section 2.4 presents implementation details, I only mention here its basic characteristics.

Although several systems have memory tags and fine-grain access control [DW89], contemporary commercial machines lack this facility. Blizzard-E synthesizes the *Invalid* state by forcing uncorrectable errors in the memory's error correcting code (ECC) via a diagnostic mode. Running the SPARC cache in write-back mode causes all cache misses to appear as cache block fills. A fill causes an uncorrectable ECC error and generates a precise exception, which the kernel vectors to the user-level handler.

The technique causes no loss of reliability. First, uncorrectable ECC faults are treated as before unless a program specified a handler for a page. Second, the ECC is only forced "bad" when a block's state is *Invalid*, and hence the block contains no useful data. Third, the Tempest library and kernel maintain fine-grain tags in user-space. The tags are used to verify that a fault should have occurred (Figure 2-3). The final possibility is that a double-bit error changes to a single-bit error, which the hardware automatically corrects. This is effectively solved by

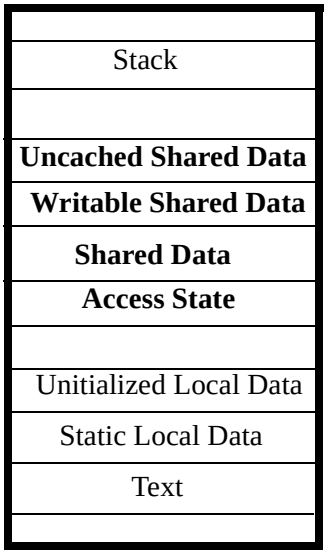

Figure 2-3. Process address space with Blizzard/E.

The figure displays the segments that comprise the address space of Blizzard processes under Blizzard/E. To support shared memory, four new segments are added to the traditional segments of a Unix process. The "shared data" segment is the address region for which shared memory is supported. The "access state" segment contains the fine-grain tags. The "writable shared data" segment is a virtual memory alias to the "shared memory segment" but the page protection for the segment pages is always ReadWrite. Similarly, the "uncached shared data" segment is a virtual memory alias to the the "shared memory segment" but accesses through this alias are not cached.

writing bad ECC in all the double-words in a memory block, so multiple single bit errors must occur.

The overhead of setting invalid ECC through an *ioctl()* system call to a special device driver is as low as 40 µsecs for an 128-byte region (Section 2.4). Resetting the ECC bits is done with uncached double words stores using a second uncached mapping on the same pages (Figure 2-3) and it does not require kernel intervention. A custom fast trap kernel interface has been implemented that reduces the round trip time per handler invocation to 4.4 µsecs, an order of magnitude faster than the round trip time of the Unix signal interface which is 101 µsecs.

Unfortunately, the ECC technique provides only an *Invalid* state. Differentiating between *ReadOnly* and *ReadWrite* is more complex. Blizzard-E emulates the read-only tag by using the TLB to enforce read-only protection. This introduces two problems. First, manipulating

the page protection requires expensive system calls. Second, if any block on a page is *ReadOnly*, the page's protection is set to read-only. Since the page granularity is larger than a cache block, writes to writable blocks on a page containing a read-only block trap to the kernel. On a write-protection fault, the kernel checks the fine-grain tags. If the block is *ReadOnly*, the fault is vectored to the user-space handler. If the block is *ReadWrite*, the kernel completes the write and resumes the computation. The kernel executes these writes within the kernel trap handler in 5.9 μsecs. When Blizzard internally stores data in the shared memory blocks, it uses a second mapping to the shared pages that always has read-write protection (Figure 2-3) to avoid the overhead of these writes.

Figure 2-3 elaborates on the structure of the application address space with Blizzard/E. On initialization, Blizzard/E maps the Blizzard ECC device driver *"/dev/eccmemd"* (Section 2.4) to the shared heap with the page protection set to none. The device driver implicitly maps the uncached and read-write aliases to the shared heap. As in Blizzard/E, the fine-grain tag table is stored just before the shared memory heap using *"/dev/zero"* with the protection set to read-write. Unlike Blizzard/S, the page protection is checked before the fine-grain tags. This does not substantially change the subsequent sequence of events compared to Blizzard/S. This time however, each page is also registered with the ECC device driver using an *ioctl()* system call to the ECC device driver.

2.3.3 Blizzard/ES: Memory Error Correction Code & Binary Rewriting

Blizzard/ES combines ECC and software checks. It uses ECC to detect the *Invalid* state for load instructions, but uses executable rewriting to perform tests before store instructions. This technique eliminates the overhead of a software test for load instructions in Blizzard/S and the overhead for stores to *ReadWrite* blocks on read-only pages in Blizzard-E. The overhead in protocol actions is higher than Blizzard/S and the access check overhead is higher than Blizzard/E.

The structure of the application address space (Figure 2-4) with Blizzard/ES is a superset of the structures with Blizzard/S and Blizzard/E. Moreover, the event sequence on the first access to a page depends on whether the access is through a load or a store and therefore, like the one in Blizzard/E or Blizzard/S respectively.

2.3.4 Blizzard/T: Custom Accelerator

The Vortex board [Pfi95,Rei96] snoops memory bus transactions and performs fine-grain access control tests. It occupies one of the two memory bus slots available in the COW nodes. It contains two FPGAs, two SRAMs, and some minor logic. The SRAM contains the fine-grain tags. The fine-grain tags are mapped into the user-address space just below the shared memory heap and can be accessed and manipulated from the user without kernel intervention.

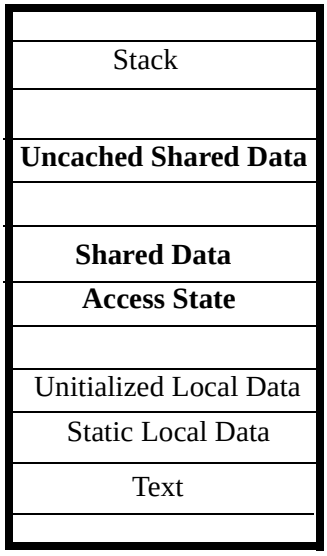

Figure 2-4. Process address space with Blizzard/ES.

The figure displays the segments that comprise the address space of Blizzard processes under Blizzard/ES. To support shared memory, three new segments are added to the traditional segments of a Unix process. The segments are the same as the corresponding ones for Blizzard/E. Unlike Blizzard/E, a writable alias to the shared memory region is not required since Blizzard/ES does not emulate the ReadOnly tag state using the page protection.

As with the ECC technique, the cache is running in writeback mode. Cache misses appear on the memory bus as bus transactions. The board monitors bus transactions and checks block requests against the fine-grain tags. If the access is not allowed, it aborts the memory operations and generates a precise exception. The offending processor receives a bus error that is trapped by the kernel. Vortex supports the *ReadOnly* state by taking advantage of the memory bus coherence protocol. When read-only blocks are loaded in the processor cache, the board imitates a processor and indicates that the block is shared. The processor notes this fact and if it attempts to modify to the block, it will first request the invalidation of any other copies. Vortex checks such requests and aborts them in the same way as the other invalid accesses.

The structure of the application address space (Figure 2-5) with Blizzard/T as well as the event sequence on the first access to a shared heap page are similar as in Blizzard/E. Blizzard/T, however, does not require a second mapping with always read-write protection since it does not use the read-only page protection to emulate the *ReadOnly* state. In addition, the Vortex device registers are mapped into the user address space [Rei96]. One of the registers is

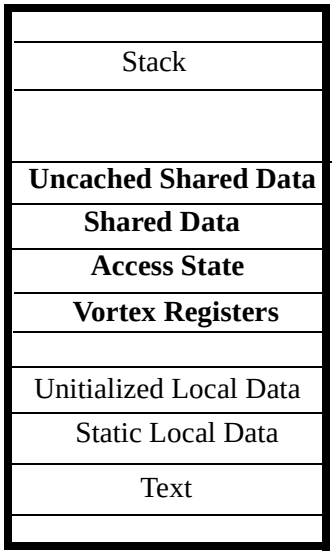

Figure 2-5. Process address space with Blizzard/T.

The figure displays the segments that comprise the address space of Blizzard processes under Blizzard/T. To support shared memory, four new segments are added to the traditional segments of a Unix process. The "shared data" segment is the address region for which shared memory is supported. The "access state" segment contains the fine-grain tags which reside on the Vortex board. The "uncached shared data" segment is a virtual memory alias to the the "shared memory segment" but accesses through this alias are not cached. The "vortex registers" segments allows user-level accesses to the registers of the vortex board.

used to facilitate access downgrades to the *ReadOnly* state. A second register is dedicated to speedup access faults. It is not used in Blizzard however, for reasons that are discussed below.

The Vortex board has been used to build a demonstration prototype of Typhoon-0. Using Blizzard's infrastructure, the Typhoon-0 prototype is an example of simple custom hardware support for Tempest. In this study, I examine Vortex's ability to offer a complete fine-grain access control solution, in isolation from its other capabilities that have been used in the Typhoon-0 prototype. More specifically, in the Typhoon designs [Rei96], there is a single computation processor while a second processor is dedicated for protocol actions. The Vortex board facilitates intranode communication by mapping to user space a cacheable control register. This register captures the physical address of the last access fault allowing the dedicated protocol processor to initiate action as soon as the violation is detected. However, it is more difficult to use this register with more than one computation processors since many processors can simultaneously cause access faults.

2.4 Kernel Support For Fine Grain Access Control

Two forms of access control in Blizzard rely on hardware mechanisms (ECC and Vortex). For both, Solaris 2.4, the operating system of the COW nodes, must provide an interface to hardware resources beyond those it was designed to support. In this section, we discuss the design and implementation of the kernel support.

2.4.1 Design Methodology

First, a constant constraint is the use of an unmodified commodity kernel. Extended functionality is offered through runtime-loadable kernel object modules. Second, a modular approach is followed. A hardware independent module extends internal kernel interfaces while hardware-specific device drivers rely on its services to support specific hardware techniques. Third, all the performance critical operations are extensively optimized.

Runtime Functionality. My goal—based on an earlier, unfavorable experience in our group with adding code to production software—was to support fine-grain access control under a standard kernel. Fortunately, Solaris 2.4 is structured in an object-oriented manner, and like other systems, supports dynamically loadable kernel modules. Much of the system is invoked through function pointers embedded in well-defined data structures. The Blizzard modules replace these function pointers to intercept and extend kernel calls. However, this approach limits the extent that the Solaris kernel can be customized because it allows only the modification of existing internal interfaces. Moreover, the portability of this approach depends on the level of the intercepted functions. Well-known published interfaces should remain portable across hardware platforms and new operating system releases. Low-level, internal kernel or hardware-specific functions, however, easily change across new operating system releases and definitely change across hardware platforms. Nevertheless, this approach is valid for a prototype system like Blizzard, where the goal is to demonstrate feasibility rather than provide an easily-maintained implementation. The wisdom of the decision to avoid kernel recompiling became clearly evident when I was able to distribute my kernel modules to other researchers that did not have access to Solaris sources. In this way, they were able to enable fine-grain access control in their environment without me distributing a replacement for the Solaris kernel that was on their machines. Subsequently, these researchers used fine-grain access control to experiment with remote subpaging policies in a network environment [BS98].

An operating system can support fine-grain access control in two ways. The first approach treats access control as a direct extension of virtual memory semantics. The fine-grain access control information is a refinement of the page protection information and each memory block can have its own fine-grain access control information. An access fault, like a protection violation, is a synchronous exception raised at an invalid access. A fault invokes a handler, which runs in the process's address space. Under this approach, the system would expose system calls to enable and disable fine-grain access control for any mapped memory region. It has clear benefits. Fine-grain access control becomes a first-class citizen within an operating sys-

tem. For example, it can be used for memory regions that represent memory mapped files or IPC regions.

However, to implement this approach, the operating system abstractions must be extended to encompass fine-grain access control. Unfortunately, virtual memory is a fundamental abstraction, so the change affects every aspect of the system. Every internal abstraction that involves page protection must be extended. Every data structure that keeps track of the page protection information must be expanded. For Solaris 2.4 in particular, among other things, this approach requires extending the segment driver interface and the hardware address translation layer to deal with extended memory attributes. In addition, the file system interfaces would be affected as well. Given the large number of fundamental changes, it would be impossible with loadable kernel modules.

Instead, a simpler approach was adopted that treats fine-grain access control as an attribute of select memory regions. The operating system supports these regions through a limited interface. The kernel treats the physical memory underlying these regions specially, by delegating responsibility to the Blizzard module. This approach was implemented with limited resources and without rewriting the kernel. The resulting fine-grain memory is efficient and provides enough functionality to support application-level distributed shared memory. An extra benefit is that it reduces interactions between the rest of the OS and the new code, which makes the development and debugging easier. It should also be noted that while the discussion in this section will focus on Solaris 2.4, the approach discussed is applicable to other operating systems (e.g., Windows NT) exactly because the interactions between the kernel and the added functionality are limited.

Modularity. The required kernel functionality has been designed following a layered approach (Figure 2-6). At the bottom layer, the Blizzard System Module (BlzSysM) encapsulates the hardware independent support. This module exports well-defined interfaces that hardware specific kernel modules can take advantage of these interfaces to expose the appropriate functionality to the user processes. Two such modules have been implemented. ECC-MemD and VortexD are the device drivers that support fine-grain access control using the ECC technique and the Vortex board respectively. This layered design avoids code duplication since much of the functionality for both hardware-specific device drivers is the same. Moreover, it facilitates the coexistence of two drivers simultaneously since intercept functions need to be installed only once.

Performance. The kernel modules attempt to reduce all the overheads associated with fine-grain access control. First, the access change overhead is reduced by avoiding kernel intervention when possible. Both the ECC and Vortex techniques require uncached aliases to the shared heap to move data to memory without triggering fine-grain faults. However, uncached aliases are not normally available to user processes. Without uncached aliases, the corresponding operations must be implemented inside the kernel, which introduces the overhead of system calls in the critical path for the operations. Second, the access fault overhead is reduced using a special fast trap dispatch interface for synchronous traps. Third, the access

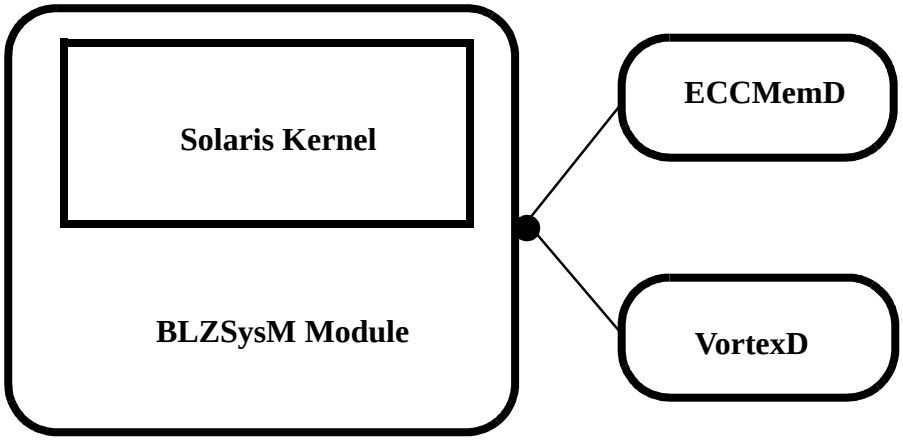

Figure 2-6. Kernel Module Structure.

The Blizzard System Module (BLZSysM) acts as a straightjacket around the Solaris kernel to extend its functionality. The ECC Memory and Vortex device drivers (ECCMemD & VortexD) support hardware fine-grain access control using its services.

check overhead is reduced for Blizzard/E by emulating writes to read-only pages within the kernel as fast as possible.

2.4.2 Solaris Basics

Solaris, as a descendant of the System V Release 4 (SVR4) Unix operating system, borrows its internal abstractions from the SVR4 virtual memory architecture[1]. It defines internal abstractions and data structures to represent an address space. The top level abstraction is the *address space* (Figure 2-7). An address space is composed of *segments*. Each segment represents an address range. For each segment, Solaris maintains a pointer to a set of functions, which are used to manage that segment. The set of functions is known as a *segment driver*. The segment driver interface is quite similar to Mach 3.0's well known external pager interface [Ope92]. Unlike Mach 3.0's interface however, it is a strictly internal interface and it does not support external pager processes.

The segment drivers maintain their own data structures for each segment. These data structures contain the protection for each page in the segment and either a pointer to the *anon_map* structure for anonymous pages (e.g., heap or stack pages) or the *vnode* structure for mem-

1. For the interested reader, an excellent source for further information is Uresh Vahalia's book titled "Unix Internals: The New Frontiers" [Vah96].

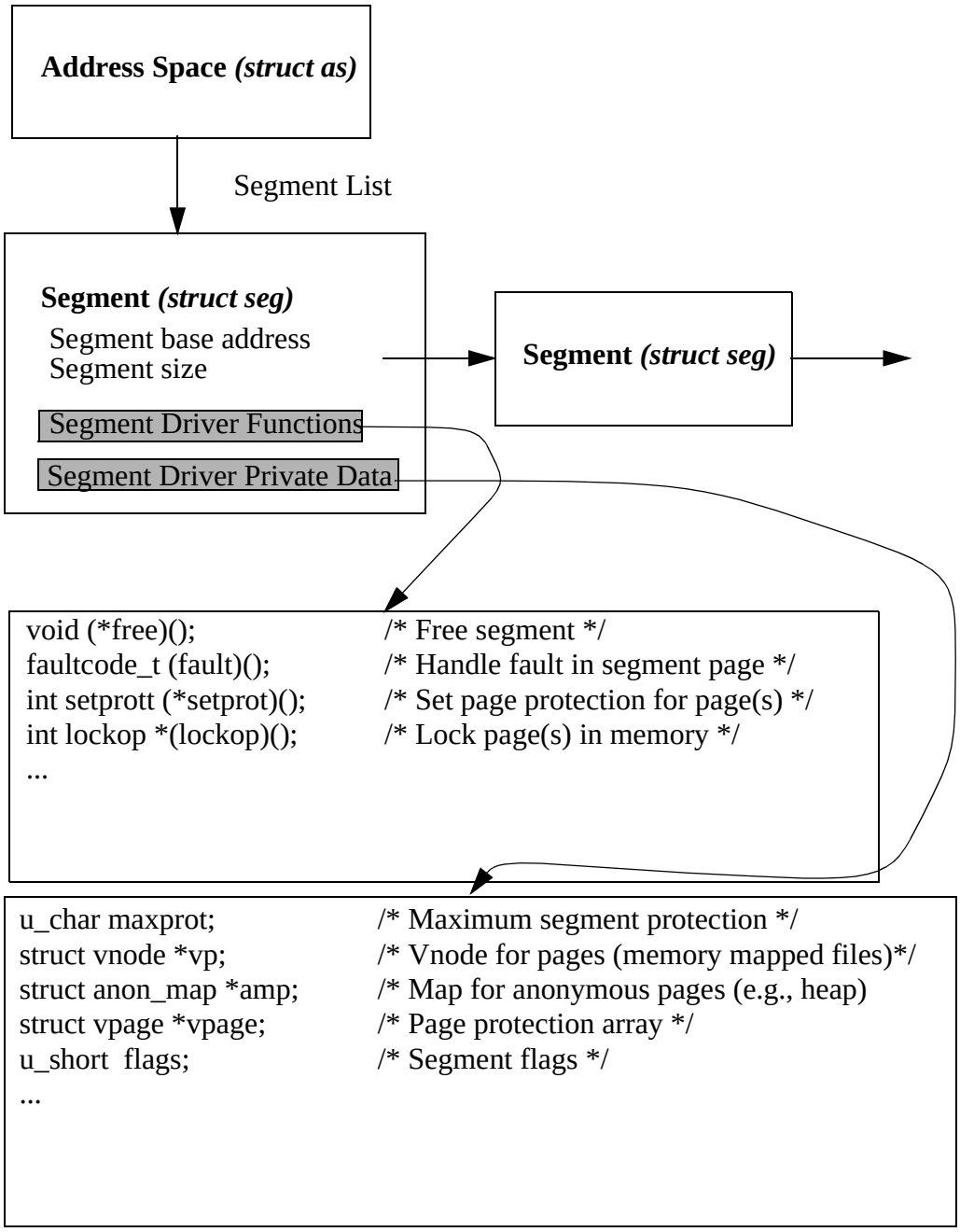

Figure 2-7. Solaris Address Space Representation.

The figure shows the data structures used in Solaris to represent an address space. Only fields important for the discussion in this chapter are listed.

ory-mapped files. Anonymous pages eventually also point to *vnode* structures of the swap filesystem. The segment driver is responsible for installing page mappings into the hardware page table in response to page faults. It consults the segment's data structures and if it determines that the access is legal loads the appropriate mappings into the hardware page table. The page mappings are installed by calling appropriate functions in the *hat* layer, which manages the hardware page table. Unfortunately, the segment driver is not notified when pages move out of the physical memory or a hardware mapping gets invalidated. The kernel already knows how to perform this operation since the *vnode* associates the page with a filesystem. Each filesystem exports appropriate functions for moving pages between the physical memory and the backing store *(fs_putpage(), fs_getpage())*. This shortcoming complicates paging for pages that have fine-grain access control enabled. Section 2.4.4 discusses such interactions in greater detail.

The BLZSysM module consists of distinct components that collaborate to modify the behavior of Solaris to support the fine-grain memory abstraction. The component list consists of a modified anonymous memory segment driver, modified high and low level kernel trap handlers, a modified *fs_putpage()* routine for the swap file system, and a modified *hat* cleanup routine. These routines are installed by substituting function pointers within the kernel to point to the module's routines and thereafter, intercept kernel calls to the original routines. The next subsections present the functionality it is offered by these intercepts. The section ends with a description of the hardware-specific device drivers.

2.4.3 Blizzard Segment Driver

The Blizzard segment driver (*segblz*) handles the shared heap and alias segments. It originated from Solaris's segment driver for anonymous memory (*segvn*). A custom segment driver is required for two reasons.

First, we must address a Solaris 2.4's limitation with regards to swap space reservation for segments that share the same physical memory. *Segvn* forces such segments to reserve in advance all the swap space they can possibly require, when the segment is first created. As we saw, Blizzard instantiates the shared heap in initialization. Given that the shared address range is typically one Gbyte, it is not desirable to reserve the swap space.

Second, we must offer extra flexibility, appropriate to support fine-grain access control. In its private data structures, the segment driver includes fields to identify the segment as one for which tagged memory has been enabled. These fields are then used by any intercept routines to determine whether to take the appropriate actions. First, it allows hardware device drivers to declare their own routines to install & delete page mappings, clean the page on segment destruction, and prepare the page when it is swapped in. Second, it allows different segments that map the same physical pages to share common private data. In this way, operations on all the different mappings to the same physical page can be synchronized.

2.4.4 Pageout Intercepts

The fine-grain access control information represents extra state that must be saved and restored when the page is swapped in and out of the physical memory, respectively. The swap data structures are not aware of the extra information. Therefore, they have not designed to handle it. Special care must be taken to perform these operations correctly. Two solutions exist to this problem. The first is to lock the pages in main memory for the duration of the program execution. The second is to modify the kernel so that it is aware of the extra information. The BLZSysM module supports both approaches allowing the device specific kernel module to choose the policy. Currently, both hardware-specific drivers lock pages in memory. The ability to run programs that require more local memory than the amount of physical memory on the COW nodes has not been very high in the priority list. However, the BLZSysM module supports paging as follows.

To deal with page-outs, the *fs_putpage()* routine of the swap filesystem has been modified to allow device specific routines for saving the access control state. As this level, the kernel does not maintain backpointers to the segment data structures. Therefore, the routine needs some other mechanism to determine whether a device specific function should be invoked. For this reason, BLZSysM maintains an array with one entry per physical page (Figure 2-8). Device specific drivers that wish to take control of the pageout activity, need to register the physical pages they manage in this array and the appropriate device specific routine will be invoked. Since the Blizzard segment driver is notified when faults occur and explicitly moves pages from the backing store to physical memory, the hardware-specific routines invoked by the segblz are sufficient to deal with page-ins. If is deemed necessary to support paging, two small functions per hardware-specific driver need to be written that can save and restore the fine-grain tags.

2.4.5 *Hat* Intercepts

The *hat* layer is responsible for managing the hardware page table. The *hat* layer intercepts are used to properly cleanup a page when the address space is destroyed and enforce the cacheability properties appropriate to implement tagged enabled memory. While the *hat* layer is the appropriate place to implement much of the functionality currently offered by the Blizzard segment driver and pageout intercepts, it is difficult to modify its behavior directly. The *hat* layer code contains references to statically declared data structures. Since these data structures cannot be accessed from outside modules, their functionality cannot be replicated.

Both hardware access control techniques use virtual memory aliases to access memory with different attributes. For example, the ECC scheme requires an uncached alias to data pages to clear invalid ECC. Vortex requires both cached and uncached aliases for its control registers as well as an uncached alias to the data pages. Unfortunately, these aliases violate the kernel's assumption that physical memory is always cacheable and device memory is always uncache-

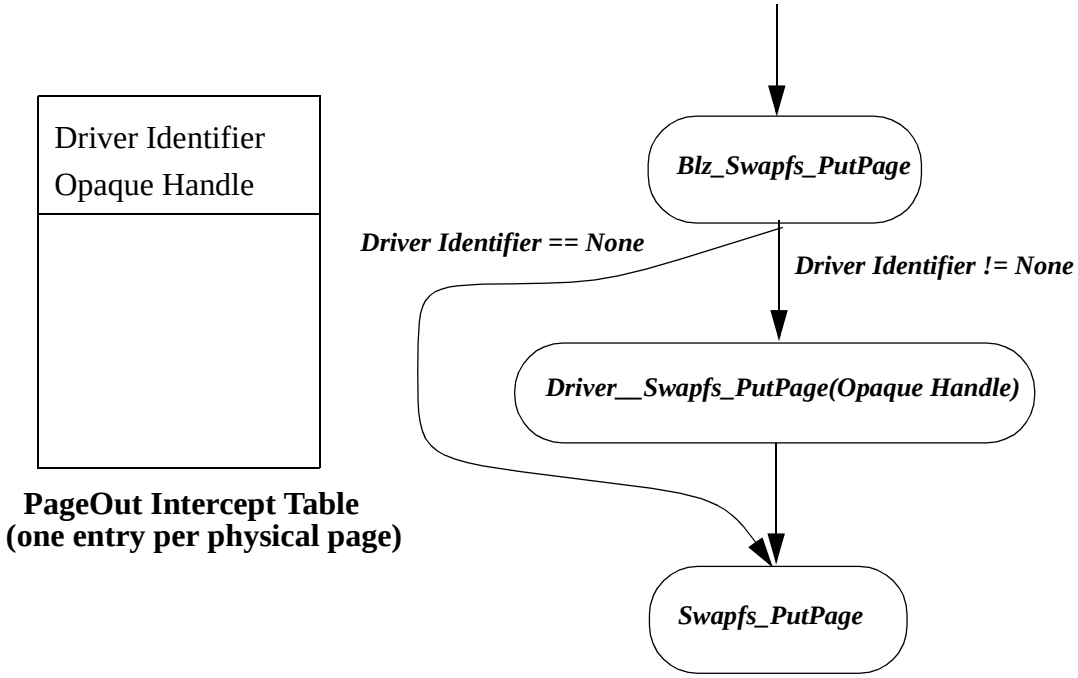

Figure 2-8. PageOut Intercept.

The BLZSysM module installs pageout intercepts so that it can notify the hardware-specific access control device drivers of pageout activity for the pages that they manage.

able[1]. To solve this problem, the module intercepts calls to the low-level (Hypersparc specific) function that modifies hardware page tables.

At each page table modification, the intercept function consults private data structures to determine the correct cacheability attribute for a page (Figure 2-9). When a hardware specific kernel module creates the segments that represent the uncached aliases, it installs appropriate entries in a table indexed by the hardware context number of the address space that installed the mapping. Whenever a page mapping is installed in the hardware page table, a lookup is performed to determine whether the address should be installed with special semantics. For the HyperSparc processor, the kernel never changes the hardware context number of a process throughout its lifetime and therefore, this method works correctly.

1. The only exception for physical memory is for processors with virtually indexed cache. In that case, if there exist two mappings for the same physical page that are not cache aligned, the kernel will force all the mappings to be uncached.

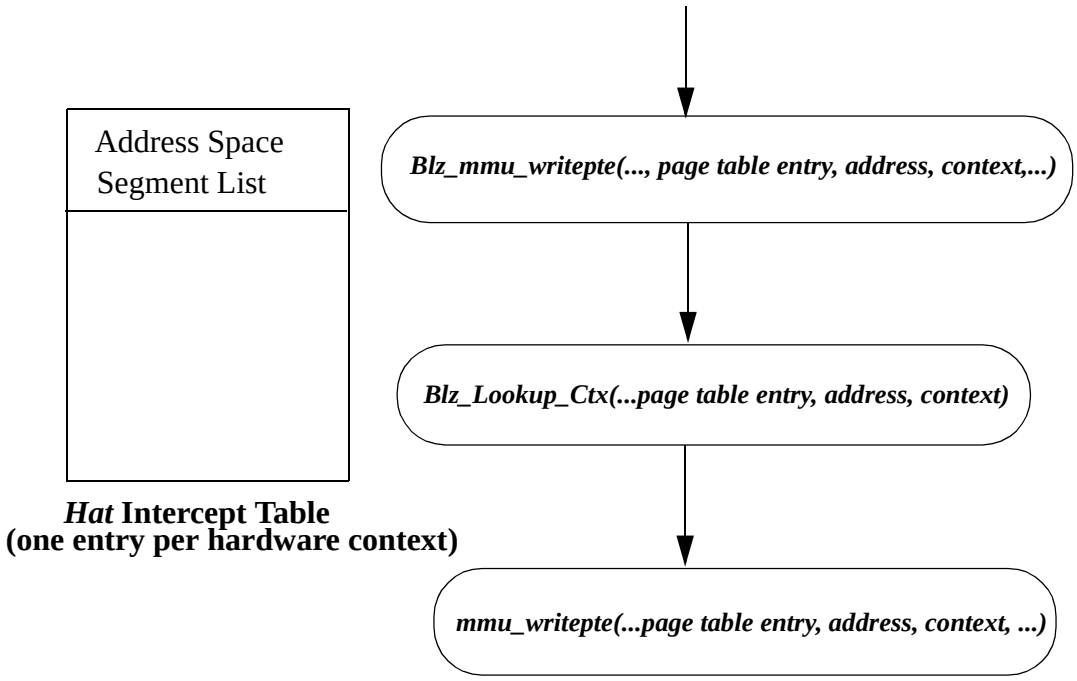

Figure 2-9. HAT Intercept.

The HAT intercepts is used to enforce the BLZSysM module's view on the page table entry properties for a page, regardless of what the kernel thinks the properties should be.

2.4.6 Low Level Trap Handler

The Solaris assembly trap handler has been modified to intercept faults. This allows altering the behavior of Solaris so that faults are delivered to the user process with minimal kernel overhead. While such functionality could have been implemented at higher layers, the speed requirement forces us to work at the lowest possible level. The modified low level assembly handlers offers two services. First, it allows quick execution of the writes to read-only pages for Blizzard/E. Second, it allows fast dispatch of synchronous traps to user-level.

The *ReadOnly* state is emulated using the page protection. The invariant is that if a block in the page is read-only the page will be mapped read-only. Therefore, writes to *ReadWrite* blocks in the same page will trap with a protection violation fault. To make the technique viable, the overhead of these faults must be very small. To achieve this, the instruction is retried inside the kernel assembly trap handler. The overhead per write is 5.81 μsecs. This specific functionality is the only place where the layered approach is not strictly followed due to per-

formance considerations that prevent the implementation of a more general interface for hardware-specific drivers.

Access control hardware generates traps that must be delivered to a user-level handler. Solaris 2.4 (like other commodity systems) is not optimized to deliver synchronous traps quickly to user-level code. Among the existing interfaces, only the signal interface can propagate traps to a user process. This interface is general and as consequence too costly for fine-grain access control. The measured time for a synchronous signal, such as SIGBUS, from a faulting instruction to the signal handler and back is 101 μsecs. In Chapter 3 we shall see that this time is longer than the roundtrip time for a message through the network. Therefore, a better solution is required to deliver traps to the user process.

To understand why the signal interface is so slow, it is worthwhile to trace the sequence of events in Solaris during a synchronous access fault (Figure 2-10 (a)). Upon encountering the fault, the CPU invokes, through a trap vector, an assembly handler in kernel mode. The handler has to save the hardware state and identify the cause of the fault. While some traps can be serviced completely at this level (e.g., register overflow/underflow traps), for most cases, a high level kernel routine needs to be invoked. This routine can access the kernel data structures and take the appropriate actions. For synchronous faults from user mode (e.g., bus errors, protection violations, etc.), the segment driver, which manages the address to which the fault occurred, will be invoked to deal with the fault. Normally, the segment driver will resolve page faults. If it fails and the user process has not blocked signals, the kernel will save the process state in the user address space. Then, it will redirect the user process to the signal handler. Returning from the signal handler, the user process will trap into the kernel mode again. Finally, the kernel will attempt to execute the faulted instruction again.

There are obvious inefficiencies in the sequence described above; too much processing is done inside the kernel, too much state is being saved before invoking the handler, a second kernel trap is used to return to the faulted instruction. The reason for this complexity is that the signal interface has been designed as a general purpose notification mechanism that deals with both synchronous and asynchronous faults.

Fortunately, there are well known techniques to support fast exceptions [RFW93, TL94a]. The main idea is that the user trap handler is invoked as soon as the kernel realizes that there is a hardware fault. In addition, the kernel is not involved again when we return to the faulted instruction (Figure 2-10 (b)). In other words, the fast trap dispatch interface reduces the handler dispatch to an involuntary procedure call relying on the trap vector code within the kernel to force the call. The kernel itself is not further involved in dealing with the fault.

Both the WWT, and later, the Blizzard/CM-5 used these techniques to speedup fault delivery. However, the CM-5 nodes run a very simple kernel without minimal support for virtual memory. Thekkath, et al. [TL94a], used similar techniques in the context of a commercial workstation operating system. Nevertheless, the operating system used, Digital Ultrix, had a

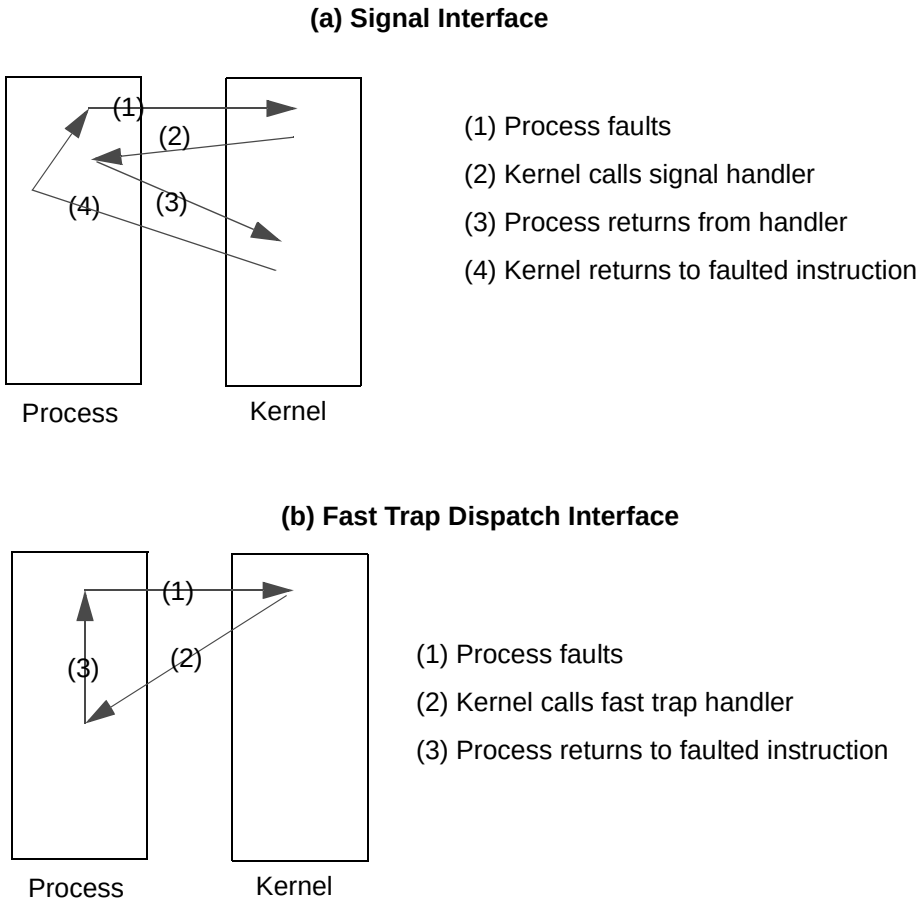

Figure 2-10. Signal vs. Fast Trap Interface.

simplevirtual memory architecture. Moreover, neither approach was designed for multiprocessor nodes, compared to Solaris.

Solaris's virtual memory architecture significantly complicates fast trap dispatch. The main problem is that it is not possible to easily access the segment data structures that describe the address space at the assembly trap handler level. Since the hardware page table is being used as a cache for translations, it may not be consistent with the kernel view. In fact, it is possible to invalidate a process's address space completely and it will be rebuilt on a demand basis as the memory access faults are serviced by the segment driver. To address this problem the fast trap interface defines best-effort semantics in the delivery of fast traps. If the kernel cannot deliver the trap using the fast trap interface, it will forward a Unix signal instead.

Best-efforts semantics also provide a solution to the following problem. The V8 Sparc architecture does not support any way to return to the user mode without involving the kernel when the *program counter (PC)* and *next program counter* (*NextPC)* are not sequential[1]. This occurs when the memory access instruction is in the delay slot of a control transfer instruction. Fortunately, there are a number of solutions. The simplest one is to use the signal interface for those faults. However, for one program that accessed shared data in tight loops, this resulted in up to 30% of the faults being serviced through the signal handler. A second solution is to install a special system call to return to the faulted instruction. However, even in this case the overhead of returning from a fault handler will be significantly high. Finally, a third solution is to simply not allow memory accesses in delay slots. Since there is a powerful binary rewriting technology used in Blizzard, it is relative easy to rewrite the executable to provide this guarantee. Therefore, this is the way that Blizzard currently deals with problem. As experiments have shown, the performance impact of this approach is minuscule. Very few memory accesses are performed in delay slots and at most the performance penalty is one cycle per such access.

To handle multiprocessor nodes, the interface defines per-thread private areas in the user address space where the fault information is being saved (Figure 2-11). Moreover, the kernel saves this information as soon as possible allowing the other processors on the same node to initiate the fault processing before even the thread which faulted realizes that a fault occurred. While the round trip time for the interface, including saving and restoring global registers in the user handler, is 4.6 μsecs, the fault information will appear in the user address space 3 μsecs after the invalid access occurred.

2.4.7 High Level Trap Handler

ECC faults or Vortex access violations cause the memory accesses to trap with an extended bus error status stored in the MMU registers. Normally when the fault is caught by the kernel, it is treated as an unrecoverable hardware error. If it occurs in kernel mode, it will result in a kernel panic. If it occurs in user mode, the process is killed and the physical page will be marked as bad. Therefore, we must install trap handlers that will treat such faults as normal access faults, when they occur in pages for which the fine-grain tags have been enabled. There are two such handlers that need to be intercepted. The first one is the basic synchronous fault bus handler. The second one is the handler used to deal with asynchronous extended bus faults. Such faults will be triggered, if an attempt is made to load data from invalid blocks using uncached memory accesses. This can occurs often enough when programs are debugged on a platform with virtually indexed caches such as the HyperSparc.

A larger complication is kernel and device accesses to shared-memory regions. The kernel typically accesses a user address space at clearly defined points. When it does, the kernel can incur access faults. The kernel deals with these faults as it handles any invalid argument and

1. This problem has been rectified in the V9 Sparc architecture.

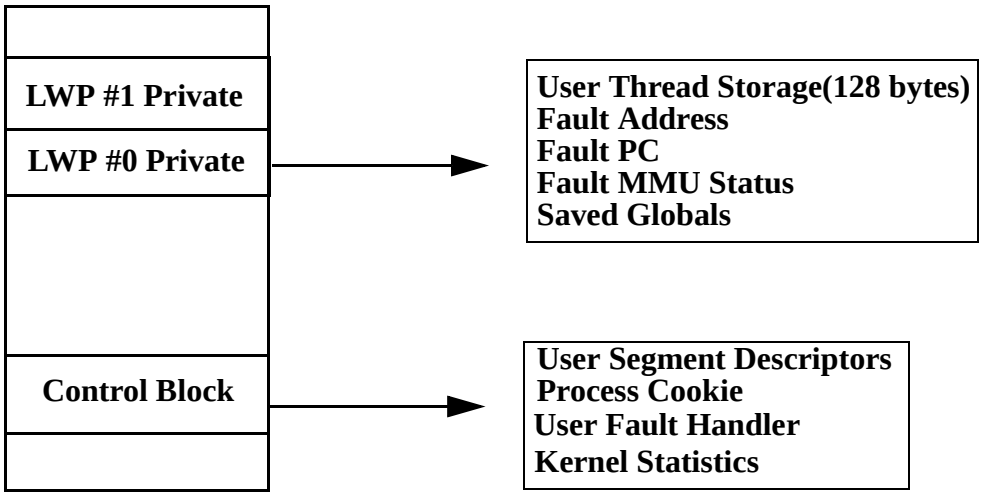

Figure 2-11. Fast Trap User/Kernel Interface.

The interface is controlled through control blocks residing in the user-address space. Control blocks represent an ad hoc extension of the traditional Unix process area, which cannot be modified through loadable kernel modules.

In Solaris terminology, kernel threads are called lightweight processes (LWPs) . Each lwp uses register %g7 to point to its private control block, which is used to save the fault information and process state. There is a single global control block for the process that: (a) describes the address range for which fine-grain access control is enabled, (b) enables the fast-trap interface and verifies that the process has requested its services (through the cookie value in the control block), (c) keeps information shared by all lwps, such as the user fault handler address, and (d) maintains fault statistics

returns an error code. A more complete alternative would invoke user handlers at these faults to make an application's fine-grain access controlled memory indistinguishable from hardware shared memory. This would require a large change, of questionable value as we can provide the same functionality by wrapping functions around system calls. A wrapper function can copy data in or out of local memory and always pass pointers in local memory to a system call. Another unresolved problem is DMA, which, for example, resets ECC bits. Since DMA to user pages only occurs when a user process directly accesses block devices, such as disk device drivers, we have not yet addressed this issue. However, a wrapper around the system calls could apply here as well.

2.4.8 Hardware Device Drivers (ECCMemD, VortexD)

The role of the hardware fine-grain access control device drivers is to implement the required functionality and present an interface to the user processes. Both drivers export appropriate functions to user processes that allow them to create Blizzard segments. Both drivers export functions that allow user programs to enable and disable fine-grain access control for a specific page in a Blizzard segment. Moreover, the VortexD device driver is also responsible for initializing the Vortex device. Since access changes completely occur at user level, it is not actively involved while the application is running. The ECCMemD driver however, has to offer an interface for accessing the memory controller.

To set invalid ECC, the memory controller must be in a diagnostic mode in which the controller does not compute ECC for subsequent writes. Instead, the ECC value comes from a register. Since this mechanism applies to all writes, all other memory traffic must stop—including other processors and DMA operations. To achieve this, the bus arbitration is disabled for every MBus master except the master processor (MBus Identifier #0). Unfortunately, the Sparcstation-20 memory bus controller cannot disable arbitration for the master processor. For this reason, the low level routine that performs this operation must be executed on this processor. If the system call originates in some other processor, a cross call must be made to master processor. To avoid this expensive operation, the protocol thread is bound to this processor, under those scheduling models that have a fixed protocol thread.

The user entry point to set invalid ECC is coded through an *ioctl()* command to the ECCMemD device driver. There are four steps involved in this operation. First, the system call arguments must be processed an the appropriate kernel structures must be accessed. Second, the validity of the request is verified. Third, processing specific to the change tag request is performed. Finally, the actual function that changes the ECC is invoked.

In Table 2.2, the overhead breakdown for the system call that sets invalid ECC is listed. In the case of the master processor, the overall overhead for 64 bytes is almost equally divided between the system call overhead, the general system call argument checking, and the command specific processing. Actually setting the invalid ECC code is a very low overhead operation that takes 3 μsecs per each 64-byte block. The per block overhead increases to 7 μsecs when the operation copies the old data out of the memory before setting the invalid ECC. Having to do a cross call from some other processor adds 30 μsecs and effectively doubles the overhead of the operation at 64 bytes.

Another performance bottleneck that the ECCMemD device driver partially addresses is the virtual memory page protection changes, which Blizzard/E uses to enforce read-only access. Changing a page’s protection, when the first read-only block is created or the last one is deleted, is costly, particularly for a multiprocessor kernel. This operation requires modifying page tables and flushing the TLBs of all the node processors. The measured overhead of the *mprotect()* system call depends on the state of the page when the protection is changed. Solaris 2.4 implements a lazy scheme for the hardware page tables. In this scheme, protection

Table 2.2: Breakdown of the time required to set invalid ECC.

Cumulative Operation overhead (μsecs)	Master Processor	Other Processors
System Call Overhead	7.3	7.3
Argument Verification	7.4	7.4
Change Tag Processing	7.3	7.3+30
Set Invalid ECC for 64 bytes (and copy out old data)	3 (6.8)	3 (6.8)

upgrades merely update the kernel data structures. Next time a fault occurs, the new mapping will be installed. On the other hand, protection downgrades require flushing the TLBs of all the node processors. The measured overhead of the call ranges from 20 μsecs (e.g., when the page has not been touched by the user process recently) to 600 μsecs (e.g., when the page protection is downgraded and a cross call is required to flush the TLBs in all the node processors). The *ioctl()* interface allows to set invalid ECC and change page protection in the same operation. This saves a second kernel entry (10-20 μsecs) per operation.

2.5 Fine-Grain Access Control Performance

There are three overhead components that determine the total overhead of the fine access control mechanism: access check, access fault, and access change. Of these three components, the most important one is the access check overhead. Indeed, since one out of five instructions is a memory operation, even a modest increase on their overhead can result in a significant slowdown. For typical shared memory coherence protocols, the other two overhead components are very closely interrelated. In such protocols, a block access fault invokes a handler that requests the data from a remote node. The remote node replies with the data and the computation is resumed. As part of coherence protocol actions, the fine-grain tags are downgraded on the remote node and upgraded on the faulted node. So, for every access fault we have more than one access change. Therefore, the change overhead is relatively more important than the fault overhead.

2.5.1 Access Check Overhead

The hardware versions, as expected, have nearly zero access check overhead. With software lookup, on the other hand, the overhead largely depends on the application. Table 2.3 presents instrumentation overhead for a collection of scientific applications. We can compare runs of

the serial program versus uniprocessor runs of the instrumented parallel program. For Blizzard/S, we observe a slowdown anywhere from 1.35-2.22 slower than the serial program. For Blizzard/ES, the slowdown is reduced and it ranges from 1.06-2.00. The most important lesson from this experiment is that the access overhead depends to the utmost degree on the application. Not only the instrumentation overhead varies between applications but also the relative overhead between Blizzard/S and Blizzard/ES is as variable. In other words, the ratio of loads vs. stores that get instrumented also depends on the application.

Table 2.3: Access check overhead for uniprocessor runs of parallel applications.

The application times are normalized against the runs of sequential applications (no FGDSM overhead). The applications and their input sets are described in Chapter 4. The Shasta results are taken from the Shasta ASPLOS paper [SGT96].

Benchmark	Blizzard/S	Blizzard/ES	Shasta without batching	Shasta with batching
Appbt	1.88	1.16	-	-
Barnes	1.63	1.22	1.14	1.08
LU	2.07	1.34	1.65	1.29
MolDyn	1.35	1.06	-	-
Tomcatv	2.22	2.00	-	-
WaterSp	1.50	1.12	1.23	1.18
WaterNq	1.59	1.20	1.22	1.14

There is evidence to suggest that the software access method can achieve better results. The evidence come from Digital's Shasta [SGT96] for which it is reported that the fine-grain access control overhead increases the application running time by 8%-40% relative to the sequential application. Shasta is an FGDSM system targeted for Digital Alpha processors that like Blizzard/S uses binary rewriting technology to implement software fine-grain access con-

trol. However, it is difficult to directly compare these results to Blizzard/S results for several reasons.

- Unlike Blizzard/S, Shasta batches accesses within the same basic block to two consecutive blocks and use a single check for all. For some applications, this optimization can reduce the overhead of fine-grain access control by more than 100%.

- Shasta is running on significantly different processor architecture. In particular, architectural factors that affect the instrumentation overhead include memory interlocks, processor caches, address space size, bit extraction instructions.

- Shasta employs commercial strength technology. Therefore, it is much more aggressive in its analysis to eliminate potential bottlenecks.

2.5.2 Access Change Overhead

In Table 2.4, we see the results from simple benchmarks that determine the access change overhead, including all the incidental overheads in the Blizzard runtime system. It includes all the possible state transitions. Since Tempest defines atomic operations to change the access state and recover or deposit a block of data, those transitions are included as well. The block size used is 128 bytes. While each Tempest implementation defines its own minimum block size, the Tempest calls can take as an argument any multiple of the basic block size. For the parallel applications that we have examined, a block size of 128 bytes works best in most cases.

At first glance, the Blizzard/E times that involve a system call catch the attention. They are an order of magnitude greater than anything else. There is large startup overhead involved with the operation. Thereafter, the overhead per block is linear to the number of blocks involved.

At the next level, we notice the Blizzard/T transitions that downgrade the block access tag. For these operations, the Vortex board must move the latest data copy to an onboard block buffer. From there, it has to be copied back to the original location using uncached stores.

Finally, for the rest, we can distinguish two classes of transitions: those that access the data and the control information and those that access the control information only. Since the control information is 1/64 of the data, the latter transitions are almost three times faster. It is interesting to note that for some transitions, the overhead of putting new data is similar to simply changing the access. In those transitions, changing the access involves filling the original block with new values. With the ECC method, this occurs when the ECC is cleared with uncached doubleword stores. With the software method, this occurs when the sentinel values are written on block invalidations.

.

Table 2.4: Access change overheads with different fine-grain tag implementations.

Operation overhead (μsecs) 128 bytes	Blizzard/S	Blizzard/E[a]	Blizzard/ES	Blizzard/T
Inv to RW	3.8	2.8	2.7	1.4
Inv to RW (put data)	3.7	2.9	2.8	3.3
Inv to RO	3.6	3.7[a]	2.7	1.5
Inv to RO (put data)	3.7	3.9[a]	2.8	2.8
RW to Inv	3.9	47.9	44.0	4.7
RW to Inv (get data)	4.5	52.7	51.4	7.7
RO to Inv	3.6	49.4[a]	48.0	4.7
RW to RO	1.3	2.3[a]	1.3	8.6
RW to RO (get data)	3.6	4.7	3.4	9.7
RO to RW	1.3	2.4[a]	1.3	1.5

a. Since Blizzard/E emulates the RO tag bit using the page protection, these operations can incur extra penalty when they operate on the first block in a page that its RO bit is set or the last block in the page that its RO tag bit is cleared. The overhead for this operation is highly variable (20-600 μsecs).

2.5.3 Access Fault Overhead

In Table 2.5, the access fault overhead is listed for a full trip to the user level handler and back to the computation. All the systems behave similar in terms of the access fault overhead. When the fault is caught by the kernel an extra ~5 μsecs is added to the trap overhead. This result is consistent with the raw round trip time of the fast trap interface for hardware access methods.

A detail worth mentioning is that the Blizzard system and the user level protocol libraries are compiled without register windows. Experiments have shown that without registers windows the compiled code is slightly slower compared to using windows when no window spiils occur. Nevertheless, using register windows results in highly variable overhead, depending on

the how many windows are available when the fault occurs. Spilling the register windows increases significantly the length of the critical path. Since there is not any way to control at which the nesting level the application code will encounter a fault, the system makes minimal use of register windows.

Table 2.5: Access fault overheads for different access types.

The experiment measures the roundtrip time to the user handler (in μsecs).

Trap RTT	Blizzard/S	Blizzard/E	Blizzard/ES	Blizzard/T
Read Inv	2.7	8.0	7.9	8.6
Write Inv	2.7	8.1	2.5	8.6
Write RO	2.5	7.3	2.5	8.6

2.5.4 Protocol Interactions

In typical shared memory coherence protocols like *Stache* (Chapter 1), we are interested in certain combinations of access transitions. These combinations correspond to the overhead imposed on the critical path for common protocol actions. In particular, we are interested in the service times of the following three common protocol operations: a read miss on an *Invalid* block, a write miss on an *Invalid* block and a write miss on a *ReadOnly* block (upgrade). In all cases, the block resides in the home node.

For a read miss, a transition for *ReadWrite* to *ReadOnly* occurs in the home node and a transition from *Invalid* to *ReadOnly* occurs in the faulted node. A data copy is required in both ends. Similarly, for a write miss on an invalid block, the corresponding transitions are, *ReadWrite* to *Invalid* and *Invalid* to *ReadWrite*, respectively. Finally, in an upgrade, the block state in the home node changes from *ReadOnly* to *Invalid* and in the faulted node from *ReadOnly* to *ReadWrite*. The values in the table also contain the overhead of the access fault. Interestingly enough, for all common Stache actions, the access control overhead on the critical path is never greater than the values listed in this table. For example, if the home node has to request the block for some other node, as soon as it gets the reply, it will forward the data, without modifying its local access state. Similarly, the invalidation of remote readonly copies occurs after a reply is sent back to the home node.

Table 2.6 lists the results from these computations. Blizzard/S is always the fastest but no single system performs worse for all three protocol actions. Therefore, it is possible that for a particular application a different system to perform better, depending on the application access

patterns. Of course, we must take into account overheads not captured in this experiment. Blizzard/S and Blizzard/ES must pay the overhead of the software checks, while Blizzard/E must pay the overhead of emulating the *ReadOnly* access state with the page protection. Chapter 4 discusses how all these factors affect the performance of parallel applications.

Table 2.6: Fine-grain access control overhead for common protocol actions.

The numbers (in μsecs) are computed by adding the overheads as computed in the previous experiments.

Protocol Action	Blizzard/S	Blizzard/E	Blizzard/ES	Blizzard/T
Read Miss	10.0	16.4	14.1	21.1
Write Miss	10.1	63.7	58.1	19.6
Upgrade	7.4	59.1	51.8	14.8

The results of this experiment allow us to characterize the performance of Blizzard's fine-grain access control mechanism relative to similar mechanisms used in other shared memory systems. The COW HyperSparc processors run at 66 MHz. Therefore, the fine-grain access control overhead on the critical path for common protocol events ranges from 500-650 cycles for the fastest fine-grain tag implementation (Blizzard/S) to 1000-4200 cycles for the slowest (Blizzard/E).

In hardware shared memory systems, the fine-grain access control overhead is an order of magnitude smaller than the best Blizzard system [K+94,LLG+92,LL97,LC96,SGC93]. Blizzard's fine-grain access control cannot reach this performance level. However, the Blizzard techniques are based on mostly commodity software and hardware.

Page-based software shared memory systems have fine-grain access control overheads that an order of magnitude greater that the slowest Blizzard system [LH89,CBZ91, KDCZ93]. Many of the kernel optimizations presented in this chapter are also applicable to these systems. As we have discussed however, manipulating the page protection is fundamentally more expensive than the fine-grain access control mechanisms used in Blizzard since all operations with software mechanisms and some operations with hardware mechanisms do not require kernel intervention. Even when the kernel must be involved, Blizzard's mechanisms are preferable with multiprocessor nodes due to the overhead of flushing the TLB of all the node pro-

cessors on page protection changes. This overhead can significantly limit the performance of page-based systems in multiprocessor nodes [ENCH96]. Chapter 5 discusses further implications of multiprocessor nodes.

2.6 Conclusions

Fine-grain access control is a fundamental operation in shared memory systems. It can be implemented in many ways that involve hardware, software or combination techniques. Blizzard supports the Tempest interface and therefore, exposes fine-grain access control as a user-modifiable memory attribute. The attribute name associated with the virtual address space and the attribute content is associated with the physical address space. Blizzard implements fine-grain access control using a variety of techniques primarily based on commodity software and hardware.

The chapter presented in detail the implementation of the fine-grain access control in four different Blizzard systems. Hardware fine-grain access control requires kernel support. Such support has been designed so that the extended functionality is implemented with runtime loadable kernel modules, follows a layered approach and optimizes performance critical operations.

Performance results indicate that the overhead of fine-grain access control in Blizzard is significantly higher than hardware shared memory but significantly lower than page-based distributed shared memory systems.

Chapter 3

Messaging Subsystem Design

Efficient messaging is important for any parallel processing using networks of workstations (NOWs). But it is especially important for fine-grain distributed shared memory (FGDSM) systems. FGDSM systems provide a shared memory abstraction in the absence of shared memory hardware. They rely on mechanisms to control shared memory accesses at a fine granularity, typically 32-128 bytes [SFL+94,SGT96]. The shared memory references can cause access control violations, which in turn trigger protocol events. During these events, a combination of message exchanges and fine-grain tag manipulations (Chapter 1) implements a shared memory coherence protocol that provides the illusion of hardware shared memory. Therefore, the performance of a FGDSM system depends both on the cost of the fine-grain access control mechanisms and the cost of messaging. In Chapter 2, we examined the cost of the access control mechanisms under different fine-grain tag implementations. In this chapter, we will focus on messaging.

Blizzard implements the Tempest interface [Rei94], which allows the implementation of shared memory coherence protocols as user-level libraries. The Tempest interface has been specifically designed to support fine-grain shared memory coherence protocols. Typically, such protocols are modeled after hardware-based invalidate directory protocols [AB86]. They transfer data in small, cache-block sized quantities. Moreover, they are blocking protocols, in which the computation is suspended on access violations until the protocol events (e.g. misses) have been processed. Therefore, Tempest requires low-overhead messages to provide low-latency communication, which is fundamental to the performance of many parallel programs.

Tempest also supports application specific coherence protocols [FLR+94]. In such protocols, the coherence protocol has been optimized to fit the application communication requirements. Often understanding the underlying communication patterns allows to transform fine grain communication to bulk data transfers. In this case, high bandwidth becomes as important as low latency. Therefore, Tempest defines an alternative messaging interface to optimize bulk data transfers, In its present form, this interface was designed by the author for the Blizzard/CM-5 system [SFL+94].

The discussion so far reveals two design requirements for Blizzard's messaging subsystem in Blizzard: low latency and high bandwidth. Commodity communication hardware and software has been successful in providing high bandwidth to applications. However, a serious impediment to building parallel platforms out of networks of workstations is the high latency and overhead of commodity equipment. Long latencies favor bulk communication, which is the antithesis of transparent shared memory's fine-grain communication. Therefore, for acceptable performance, Blizzard requires a network layer with moderately low latency.

Fortunately, FGDSM systems are not the only possible beneficiary from low latency communication. During the last few years, it has been acknowledged that low latencies are important for client/server or clustered applications [ACP94,ACP95,Bel96]. Consequently, novel hardware and software designs sought to address this problem. The result has been novel research and commercial designs that often achieve a fraction of the latencies of the traditional communication architectures [CLMY96].

On the hardware side, the overhead of traditional, kernel-arbitrated access to a network has been too large for low latency communication. However, a network device's memory can be mapped to the user address space. This approach permits a program to communicate without kernel intervention and is supported in recent low-latency communication hardware [BCF+95,BLA+94, Hor95,OZH+96].

Fast access to the hardware by itself is not enough to achieve low latencies. As the experience from message-passing multicomputers has shown [vECGS92, PLC95, vEBBV95], traditional messaging interfaces have not been able to realize the hardware performance due to fixed software overheads associated with sending and receiving messages. Therefore, a newer generation of low overhead messaging interfaces attacked the software overheads. Some software architectures originated in the networking community [DP93,Osb94] while others arose from the multicomputer community [vECGS92,PLC95, HGDG94]. The emergence of system area networks and networks of workstations have blurred the distinction. In general however, the latter have been more preoccupied with low latencies than the former.

Among the key proposals that emerged from the multicomputer community have been the Berkeley "active" messages. The design has been heavily influenced by earlier work in message-directed computation in the context of dataflow architectures and the J-machine [DCF+89, PC90]. Berkeley active messages sought to reduce latencies by eliminating the software complexity associated with traditional multicomputer messaging interfaces. Tempest's

messaging interface is based on the Berkeley active messages [vECGS92]. It differs from Berkeley active messages in two important aspects that are necessary to use Tempest messages to implement *transparent* shared memory protocols. First, Tempest messages are not constrained to follow a request-reply protocol. Second, Tempest messages are delivered without explicit polling by application programs.

The Wisconsin Cluster of Workstations (COW) includes low-latency Myricom's Myrinet hardware [BCF+95]. The Blizzard messaging subsystem originated out of Berkeley's Active Messages (AM) messaging library[1] [CLMY96]. By carefully considering hardware-software tradeoffs, such as interrupts or polling upon message arrival, it achieves latencies as low as 35 μsecs round-trip for control messages or 72 μsecs for data messages with 128 byte data blocks (Section 3.7).

The contributions of this study are twofold. First, it presents a multithreaded, high performance network subsystem designed to support low latency and high bandwidth communication for FGDSM systems. Second, it demonstrates that the functionality of the Berkeley active messages can be extended to expand the range of user protocols that it can support and free the user of the considerations of explicit polling without adverse performance effects.

The rest of the chapter is organized as follows. Section 3.1 reviews the Tempest messaging interface and discusses its implications for the design of the low level network protocols. Section 3.2 discusses the buffer allocation problem and develops a methodology to determine an upper limit on the buffer requirements of parallel programs. Section 3.3 describes and evaluates different buffer allocation policies. Section 3.4 presents the COW hardware and software communication infrastructure. Section 3.5 discusses the problem of transparent message notification. Section 3.6 describes implementation details of the Blizzard messaging subsystem. Section 3.7 presents microbenchmark results to characterize Blizzard's messaging performance. Finally, Section 3.8 summarizes with the conclusions of this study.

1. During the early stages in Blizzard's development, the Illinois Fast Messages (FM) library [PLC95] was used as the low-level communication infrastructure. We switched to the Berkeley library for two reasons. First, licensing restrictions made it very difficult to customize the FM library for Blizzard. Second, FM's performance was modestly worse than AM's for typical shared memory coherence protocols. While with the FM library, the latency for messages with data blocks was slightly lower than with the AM library, the latency for small messages without data blocks was significantly higher (see Section 3.1 for the distinction between the two types). In shared memory coherence protocols, messages that do not contain data blocks outnumber messages that do.

3.1 Tempest Messaging Subsystem

In this section, we will review Tempest's messaging interface. We will concentrate on its differences for other similar interfaces. Furthermore, we will discuss the complications that it introduces in the design and implementation of the low level network protocols.

3.1.1 Fine-grain communication: Active Messages

The programming model supported by Tempest is based on the Berkeley active messages [vECGS92]. A processor sends a message by specifying the destination node, handler address, and a string of arguments. The message's arrival at its destination creates a thread that runs the message handler, which extracts the remainder of the message.

As in Berkeley active messages, the Tempest message handlers are atomic and run to completion without blocking. In Berkeley active messages, the justification for atomic handlers was the high costs associated with blocking handlers. Previous message-driven architectures such the J-Machine and Moonsoon [DCF+89,PC90] had supported blocking handlers at a high cost, either in processing overhead or hardware resources [vECGS92]. For example, in J-Machine when a handler was blocked, the message had to be copied in the node's local memory to allow further processing of messages. Therefore, Berkeley active message sought to eliminate such costs using atomic handlers.

However, atomic handlers are not an ideal solution when symmetric multiprocessor workstations are used as building blocks for NOWs. By definition, only one handler can be executing at any particular moment and this limits the available processing bandwidth. An alternative approach relies on local memory-based locks to synchronize protocol handlers. This allows concurrent execution of protocol handlers, which increases the available processing bandwidth. However, locks should only be allowed to support mutual exclusion. Condition synchronization, in which the synchronization primitives are used to arbitrate the execution order [AS83], should not be allowed. In practical terms the meaning of this restriction is that the handlers should release all locks held before they complete. This model is sufficient for shared memory coherence protocols that only require synchronization mechanisms to safely access internal protocol data structures. Currently, Blizzard does not use protocol locks because they do not offer any performance advantage on the COW platform. This design choice is further explained in Section 5.2.

Figure 3-1 lists the Tempest active message primitives. Tempest supports two variants of message primitives to initiate the transfer of small and large messages respectively. Primitives that send small messages accept only word arguments. Primitives that send large messages accept a memory block specifier (start address and length) in addition to the word arguments. The memory block specifier can refer to a memory block (cache-block aligned) or a memory region (word aligned). The former are optimized for transferring cache blocks between nodes and among their arguments include the tag change for the memory block that should occur atomically with the message injection. The latter are used for more general message opera-

Send Primitives

```
void TPPI_send_{W*}(dest, function, w1,..., wn)

void TPPI_send_{W*}R(dest, function, w1,..., wn,
                     region address, region length)

void TPPI_send_{W*}Ba(dest, function, w1,..., wn,
                     block address, block length)

void TPPI_send_{W*}Rs(dest, function, w1,..., wn,
          region address, region length, remote region address)

void TPPI_send_{W*}Bs(dest, function, w1,..., wn,
          block address, block length, remote block address)
```

Receive Primitives

```
handler function(src, message size) /* handler invocation */

int TPPI_recv_W()

void TPPI_recv_R(region address)

void TPPI_recv_Ba(block address)
```

Figure 3-1. Tempest Active Message Primitives.

Tempest send message primitives can send one or more words (W) and zero or more regions(R, Rs) or blocks (Ba, Bs). Blizzard only implements primitives that send a single region or block. More primitives can be implemented if the needs ever arises. With {*}Rs and {*}Bs primitives, the user can specify the destination address for the region or block data. Tempest receive message primitives are used within the handler that is invoked upon message arrival. They can receive the corresponding words, regions or blocks contained in the message.*

tions on non-aligned regions. With large messages, the sender can also specify the destination virtual address in the receiver's address space where the data will be deposited. This allows the network subsystem to attempt optimizations such as the elimination of redundant data copying (Chapter 6). Once the handler is invoked, it has to explicitly pull the message out of the network. If the memory block's sender specifies a destination virtual address, the primitive to receive message blocks is treated as a null operation.

Tempest active messages break away from the tradition of the Berkeley active messages and derivative designs in two important aspects that are necessary for using Tempest messages to efficiently implement *transparent* shared memory protocols.

The first difference is that messages are not constrained to follow a request-reply protocol. In the Berkeley active message terminology [vECGS92], messages are distinguished into two classes. The first class contains messages injected into the network by the computation and are called request messages. The second class contains messages injected into the network by the active messages handlers. Such messages are only allowed as a reply to the node that generated the active message handler and therefore, they are called reply messages. However, it is inefficient to implement shared memory coherence protocols with pure request/reply semantics. For example, a common protocol operation is for a block's home node to forward a request for the block, on behalf of a third node, to the node that has the exclusive copy. With a pure request-reply protocol, the operation would have to split in two phases; one to contact the home node and get the current block owner, and one to contact the owner requesting the block. In general, pure request/reply protocols can result in twice as many messages as equivalent protocols without such restrictions. Section 3.2 and Section 3.3 elaborate on how this issue can be addressed.

The second difference is that messages are delivered without explicit polling by an application program. Tempest message handlers must appear interrupt driven, since they run in a software layer below an application program, which is unaware of the underlying communication and can hardly be expected to poll for it. Explicit polling has been a afterthought in active message implementations, introduced to overcome the high cost of hardware interrupts and provide a cheap mechanism to implement critical sections. In the FGDSM context, however, explicit polling is not a solution. Moreover, Blizzard offers its own set of sychronization primitives that do not rely in explicit control of the messaging activity. Therefore, Blizzard needs to rely in other mechanisms to service incoming messages. Such mechanisms are discusses in Section 3.5.

3.1.2 Bulk-data transfers: Channels

For efficient transfers of large quantities of data, Tempest includes a bulk-data transfer interface. The interface has been roughly modeled from operating system interfaces used to manage direct-memory-access (DMA) controllers. The processors initiate a bulk data transfer by specifying virtual addresses on both source and destination nodes. Figure 3-2 lists the calls associated with bulk data transfers.

The sequence of events is as follows. First, the sender requests the allocation of transfer channel specifying the source address and length, the destination node and a completion function using the primitive *TPPI_set_channel_src()*. The primitive returns a transfer channel identifier that must be communicated to the receiver using the active message primitives. The receiver completes the channel setup in its own end, by specifying the destination address, the transfer length and the completion function using the primitive *TPPI_set_channel_dst()*. When the transfer, which is initiated with the primitive *TPPI_channel_send()* ends, the nodes can either destroy the channel using the primitives *TPPI_destroy_channel_src()* and

```
int TPPI_set_channel_src(dest, send completion function)

void TPPI_set_channel_dst(src, channel, buffer address,
          buffer length, receive completion function)

void TPPI_channel_send(dest, channel,
          buffer address, buffer length)

void TPPI_destroy_channel_src(dest, channel)

void TPPI_destroy_channel_dst(src, channel)

void TPPI_reset_channel_dst(dest, channel)

void TPPI_reset_channel_src(src, channel)
```

Figure 3-2. Tempest Bulk Data Transfer Primitives.

The sequence of operations to setup and use a message channels is described in the text. Tempest offers primitives to create, send data, reset or destroy a channel.

TPPI_destroy_channel_dst(), or reset the channel to allow further transfers using the same delivery buffer using the primitives *TPPI_reset_channel_src()* and *TPPI_reset_channel_dst()*.

The Tempest channel interface was first developed for the Blizzard/CM-5 [SFL+94]. On the CM-5 platform, the small network packet size (5 words) supported by the network hardware did not allow the efficient implementation of active messages with large data blocks. Therefore, the cost of breaking a bulk data transfer into small active messages was too high to realize bandwidth comparable to the one achieved by the default CM-5 messaging library [Thi91]. In contrast, on the COW, the use of the channel interface has been depreciated mainly because the active message layer can support messages up to four Kbytes, which are big enough to offer the raw hardware thorughput. While the channel interface is still supported in Blizzard, it is being implemented using active messages. Nevertheless, it remains part of the Tempest specification, since it is capable of abstracting hardware capabilities that accelerate bulk data transfers such as DMA engines[BCM94].

3.2 Buffer Allocation And Active Messages

Dealing with buffer allocation is a fundamental problem in any messaging subsystem. An incoming message requires temporary buffers where it is held until it can be processed. Therefore, a policy is required either to guarantee that available buffers exist or to take corrective actions when no such buffers are available.

Traditional network abstractions such as UDP/IP sockets [PD97] address the buffer allocation problem by simply dropping messages when the network buffers fill up. These protocols rely on the existence of the standard mechanisms in place for error detection and retransmission to recover from such events. However, this approach is not appropriate for low latency fine-grain communication in system area or multicomputer networks where often no retransmission mechanisms are assumed since they introduce extra overheads. Therefore, protocols designed for such environments favor solutions that guarantee buffer availability.

Traditional multicomputer messaging models address the buffer allocation problem with a variety of techniques [vE93]. The data transfer either occurs after both the sender and the receiver have called the appropriate primitives (mostly used with sychronous messaging primitives) or the messaging subsystem allocates buffers on the fly as soon the data reach the destination (mostly used with asynchronous messaging primitives). However, studies have shown [vECGS92] that buffer management overheads, regardless of the kind of primitives used, often dominate the communication cost. It is not unusual for the traditional message models to achieve a fraction of the raw hardware performance.

The designers of the Berkeley active messages argued that the problem with previous approaches was in the “semantic gap” between the messaging model and the communication hardware architecture. Attempting to support a model far removed from the hardware prohibited the realization of the full hardware potential, especially for fine-grain communication where low latencies are critical.

The situation was conceptually similar to the one that computer architects had faced earlier with complex instruction sets. Such instruction sets required extensive logic to decode and execute the instructions which introduced significant overheads even for simple operations [HP90]. The “semantic gap” between the instruction set and what the hardware could directly support with simple instructions limited the performance of microprocessors as they had to rely in slow complex mechanisms to implement the instructions. This problem was addressed with the introduction of “reduced instruction set computers” (RISC). The RISC instruction set exposed simple instructions that the hardware could efficiently implement while the complex functionality was implemented in software using a sequence of simple instrutions.

Accordingly, Berkeley active messages applied the RISC lessons in the design of the messaging subsystem and its programming abstraction [vE93]. Following this design philosophy to its extremes, user protocols were restricted to pure request/reply operations, with the rationale being to reduce the buffer management overheads associated with more permissive messaging models. However, it can be argued that the push to simplicity went further than it should. If the user protocol does not naturally adhere to pure request/reply semantics, then there is a real performance penalty since up to twice as many messages may be required for an equivalent pure request/reply protocol. Nonetheless, messaging operations are significantly more expensive than local processing that may be required to support more general operations. As we shall in this study, it is possible to implement active messages without restricting the programming model to pure request/reply protocols. Moreover, a large class of general-

ized protocols can be supported as efficiently as pure request/reply protocols, including many typical shared memory coherence protocols.

The rest of this section is organized as follows. First, I introduce a methodology useful to analyze parallel programs and understand their buffer requirements. Then, I identify the important characteristics of user protocols in terms of their buffer requirements.

3.2.1 A Methodology For Analyzing Buffer Requirements

Different parallel programs produce different messaging patterns. If we know the messaging patterns of a particular program, we can readily compute its buffering requirements. However, the number for a particular program is not significant by itself in understanding the buffer allocation problem. Instead, it is more useful to determine an upper limit to the program's buffer requirements. The basic idea is to determine the maximum number of messages that can possibly remain unprocessed at any particular instance during the execution of the program. These messages must be buffered by the messaging subsystem. Knowing an upper limit to the buffer requirements will subsequently allow us to link different programming styles to their buffer requirements. Eventually, we can use this knowledge to evaluate different buffer allocation policies for particular classes of applications. Throughout this section, I assume a generalized active message model that, unlike Berkeley active messages, is not restricted to pure request/reply semantics.

First, I will introduce the concept of the message activation graph. We define the relation *'generates'* as follows. A message M_i generates a message M_j if and only if the handler invoked to receive M_i sends message M_j. This relation is transitive and therefore, a message M_i can generate a message M_j by first generating an intermediate message M_k that in turn will generate message M_j. We can now define the message activation graph of a message M_i as the transitive closure of this relation, as the set of all the messages that are generated directly or indirectly by that message.

The definition of the relation *'generates'* also allows us to distinguish between two classes of messages. The first class contains the messages that are injected into the network by the computation. The injection can be direct using the send primitives or indirect through access faults[1]. The second class contains the messages that are sent from within the message handlers. We will refer to the first class as *injected messages* and to the second class as *generated messages* since they are generated by other messages.

1. Access fault handlers are specific to FGDSM systems. While an access fault handler has the same synchronization requirements as a message handler (it executes to completion and is atomic with respect to message handlers), invoking such an handler does not consume network buffers since they are scheduled through the access fault queue. Nevertheless, for the purpose of this analysis, they will be treated as message handlers to simplify the presentation.

This classification is important because there is a key distinction between the two classes of messages. In particular, generated messages impose a constraint on how the message subsystem can deal with buffer shortages. If no buffers are available when a message is generated within a handler, the system cannot process pending messages to relieve the resource shortage. Processing any pending messages would require a handler to be launched that would execute concurrently to the one that generated the message. In active messages, where by definition a message handler executes atomically with respect to other handlers, the problem is further exaggerated. However, it is not restricted only to pure active messages; more general message-driven models that restrict this limitation have similar problems [WHJ+95]. Each blocked handler requires a distinct thread of control with its associated resources. We cannot expect the messaging subsystem to efficiently support an infinite number of concurrent handlers.

Using these definitions, we can now develop a methodology for determining the buffer requirements of parallel programs. To introduce the concepts, we assume a single node that sends active messages to itself. The process starts with the injection of a message. When its handler is invoked, it will generate more messages that in turn will cause their handler to be invoked. Since a single handler is executing at any particular moment, the active message layer must temporarily store the newly created messages until the time comes to process them. If we assume a FIFO processing order for the messages, the whole process is equivalent to a breadth-first traversal of the message activation tree. We can see an example of this process in Figure 3-3. If $child(H_{(l,i)})$ is the number of messages generated by handler $H_{l,i}$ running *i'th* among the handlers at level *l*, the stack length and therefore, the number of required buffers while executing handler $H_{l,i}$ is:

$$RequiredBuffers(H_{l,i}) = \sum_{k \le i} child(H_{l,k}) + 1 + \sum_{m > i} child(H_{l,m})$$

Therefore, the buffer requirements for the entire activation graph is the maximum value attained by this expression during the execution of the graph.

In order to generalize the methodology for many nodes, it suffices to partition the message activation graph in distinct subgraphs. The partition method depends on the characteristics of the buffer allocation policy. In particular, we can classify such policies according to whether receivers maintain a common buffer pool for all senders or distinct buffer pools for each potential sender. If there are distinct buffer pools, messages from different senders do not compete for buffers at the receiver node and we must consider buffer requirements for every combination of sender and receiver nodes separately. Therefore, each subgraph should contain the messages for each pair of sender and receiver nodes. If there is a shared buffer pool, all senders compete for buffers at the receiver node. Therefore, each subgraph should contain the messages sent to a receiver by any sender. Figure 3-4 shows an example of this process.

Finally, the methodology can be generalized for many message activation graphs being processed by the messaging subsystem concurrently; it is sufficient simply to add the buffer

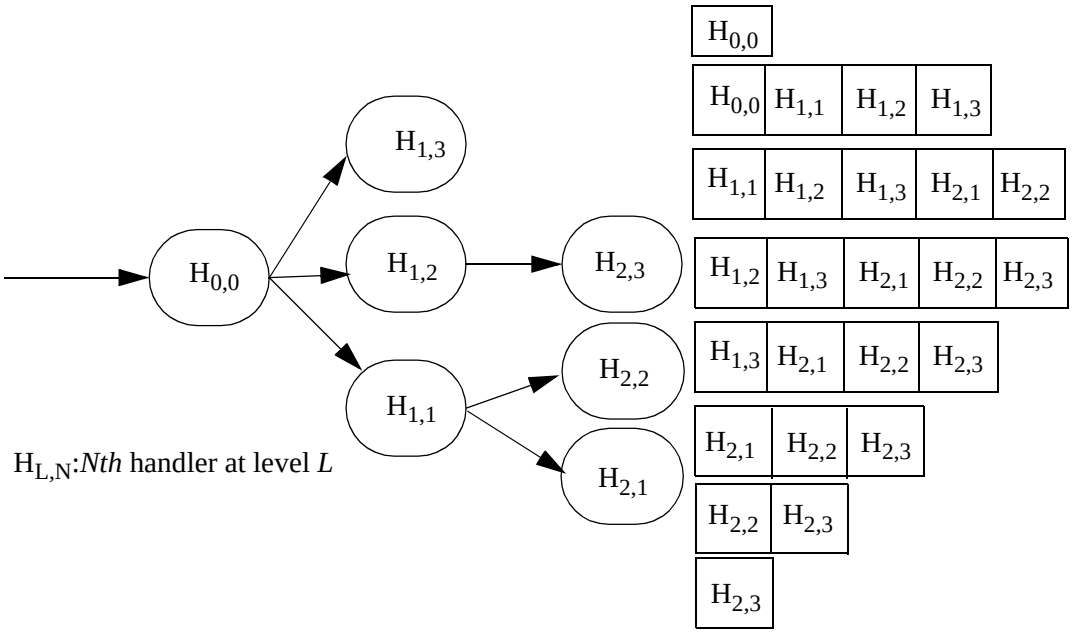

Figure 3-3. Example of a message handler activation graph with a single node.

On the left side, we see the message activation graph with the handlers represented as vertices and the message as edges. On the right, we see the queue contents as we do a breadth first traversal of the graph. This is equivalent to the buffer requirements when executing the message handlers in a FIFO order.

requirements for each graph. In this way, we determine an upper limit to the buffer requirements of a specific parallel program.

3.2.2 Parallel Programs and Buffer Requirements

We can now characterize the pressure that different parallel programming models or styles impose on the buffer allocation policies of the messaging subsystem. This characterization will prove useful in Section 3.3 for determining the strengths and weaknesses of different buffer allocation policies.

To approach the problem, we partition the messages generated throughout the execution of a parallel program into the message activation graphs of the injected messages. Then, we partition these graphs by extracting the subgraphs for each pair of sender/receiver or each receiver depending on whether the receiver maintains distinct buffer pools for each sender or there is a common buffer pool for all senders. Subsequently, we focus in the characteristics of this partition and in particular, in its two characteristics that are important with regards to the buffer

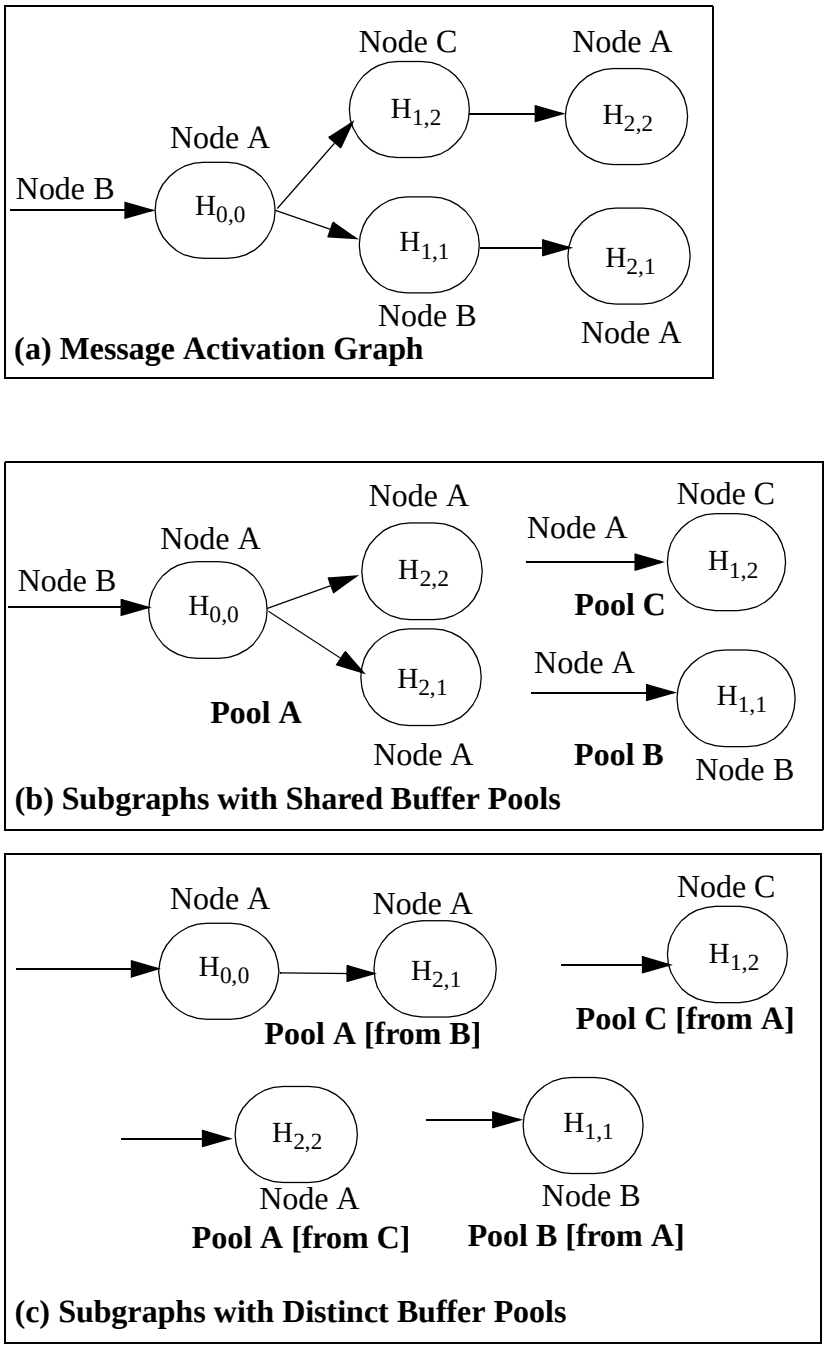

Figure 3-4. Partitioning a message activation graph with three nodes.

(a) presents the message activation graph for a message injected by Node B. With shared buffer pools for all the senders, we partition the graph in the subgraphs as shown in (b). With distinct buffer pool for each sender, we partition the graph as shown in (c).

allocation policies. The first characteristic is the buffer requirements of the resulting subgraphs compared to the number of the available buffers. The second characteristic is the number of the activation graphs that are being concurrently processed. These two characteristics define two orthogonal dimensions along which we can classify parallel programs and the user messaging protocols that they employ. Table 3.1 lists popular parallel programming models according to the characteristics of their message activation graphs.

Table 3.1: A classification of user protocol in terms of buffer requirements.

Buffer Requirements Per Graph	**Concurrent Activation Graphs**	
	Few	**Many**
Small	Invalidate DSM coherence protocols [LH89, RLW94, SGT96]	Update DSM coherence protocols, application-specific protocols [FLR+94]
Large	Dataflow protocols [CCBS95]	

Streaming protocols generate many concurrent activations graphs with small buffer requirements. Examples of such protocols include many protocols developed on top of active messages using Split-C [CDG+93,MVCA97] as well as update-based application-specific coherence protocols developed for Tempest [FLR+94]. The common characteristic of these protocols is that there is not an inherent limit in the number of injected messaged. Instead, they rely on the messaging subsystem to implicitly limit the injection rate.

Request protocols generate few concurrent requests which most of the time have small buffer requirements. Such protocols are limited by design on how many messages can inject into the network. Invalidate-based shared memory coherence protocols [LH89, RLW94, SGT96] including Stache, belong in this category.

Shared memory programs using Stache can inject at most as many messages as the number of block access faults that can occur concurrently. This number is limited by the number of threads in the program. Moreover, the maximum buffer requirements for any Stache protocol event is not ever higher than one, with distinct buffer pools per sender, or the number of nodes in the system, with a shared buffer pool for all senders. Therefore, if there at least as many buffers for every sender as the number of threads in the system, then shared memory programs will never require more than the available buffers. More precisely, if T is the number of threads per node and N is the number of nodes, then each node must have at least $N*T$ buffers available in every node to accommodate the maximum buffer requirements. Typically, designers of hardware shared memory coherence protocols know the buffer requirements for their protocol and therefore, they make sure that the network has enough buffers available to avoid

deadlocks [WGH+97]. However, such strategy is not appropriate in a general platform like Blizzard, which it is targeted to allow the development of application specific protocols.

Finally, there are protocols that create an arbitrary number of concurrent activation graphs with large buffer requirements. Such protocols often resulted from the compilation of programs written in languages with implicit dynamic parallelism (Multilisp, Id) into a dataflow graph when targeted to message-driven architectures [Hal85,Nik94]. Depending on the parallelism available dataflow graphs with different characteristics are generated. Similar protocols appeared for the execution of irregular directed acyclic graphs in distributed memory machines that were developed for numerical computations [CCBS95].

The classification of parallel programming models reveals that the messaging subsystem must provide three kinds of services to user protocols:

- It must be able to accommodate and service short-term bursts.
- It must provide long-term flow control for those protocols that require a feedback mechanism.
- It must be robust enough to handle protocols with increased buffer requirements beyond the available buffer, presumably at a reduced performance level.

3.3 Buffer Allocation Policies

Network interfaces (NIs) usually provide onboard buffers to receive incoming messages. A larger number of available buffers allows programs with higher buffer requirements to be executed without special actions from the network subsystem. However, it is always possible to construct a program that requires more buffers than the ones available. For this reason, the messaging subsystem must be able to deal with such shortages using an appropriate policy. In this section, we will review buffer allocation policies, discuss their strengths and weaknesses for FGDSM systems and contrast them to the one designed for Blizzard.

We will examine four basic policies: *pure request/reply, return-to-sender, sender-overflow* and *receiver-overflow.* These policies differ on how they deal when the buffer requirements of a user protocol exceed the available buffers. The *pure request/reply* policy effectively solves the problem away by restricting the allowed user protocols. *Return-to-sender* buffer messages in the local memory of the sender after they have been bounced back by the receiver when buffers are not available. *Sender-overflow* also buffer messages in the local memory of the sender but this time it uses a flow control mechanism at the sender to keep track of the available buffers on the receiver. Finally, *receiver-overflow* buffers messages in the local memory of the receiver when possible deadlocks are detected.

Traditional network protocols either share the available buffers among all the senders for connectionless protocols such as UDP or support only one sender for connection-oriented protocols such as TCP. In contrast, low level network protocols for active messages typically allocate distinct buffer pools for each potential sender. Such protocols use credit-based flow control schemes to pipeline the injection of new messages with the receipt of acknowledg-

ments. These schemes are very similar to flow control schemes originally used in link-level flow control protocols for virtual circuits [PD97]. The only exception to this rule is *return-to-sende*r which uses shared buffers.

3.3.1 Pure Request/Reply

Berkeley active messages [vECGS92] dictate that the application should be limited to pure request/reply message operations. In this way, the programming model itself forces the parallel programs to reduce the size of the message activation graphs for injected messages. With pure request/reply protocols, the graph for each injected message contains exactly two messages, the request and the reply. Therefore, each message exchange requires exactly one buffer at each end of the communication.

The advantage of this constraint is that it allows a very simple buffer allocation policy [Mar94] based on a simple credit-based flow-control scheme with distinct buffer pools for each sender. Half of the available buffers on each node are allocated for requests and half for replies. Then, the system implicitly preallocates a buffer for the reply when a request is being sent. A node must always send replies, which acknowledge the corresponding request and indicate that its buffer has been freed.

This policy effectively provides flow control for user protocols that can naturally be expressed as *pure request/reply* protocols. However, the restriction in the programming model can be cumbersome if the user protocol is not such protocol. As we have already discussed, shared memory coherence protocols cannot be efficiently implemented[1]. Therefore, it is inappropriate for Blizzard. However, its simplicity results in efficient implementations and hence in good performance, for those user protocols that it is applicable. Consequently, it can serve as the benchmark against which we can judge more sophisticated policies.

3.3.2 Return-to-Sender

This policy briefly appeared in the first release of the Illinois Fast Messages (FM) for Myrinet [PLC95]. It was inspired from the deflection and chaos routing policies of the Tera-1 machine [ACC+90]. This policy uses a shared buffer pool for all senders. It is based in the idea that messages which cannot be delivered to the receiver will be routed back to the sender. There, they will be placed in an overflow queue and retransmitted again. Messages in the overflow queue have priority over newly generated messages. Therefore, if a nodes attempts to send a message, it first has to make sure that the overflow queue is empty.

1. In reality the situation is even worse. It is impossible to use this policy in Blizzard since Tempest access fault handlers execute atomically with respect to message handlers. Typically, when an access fault occurs, the access fault handler sends a request to the home node. A deadlock can occur at this point since new messages cannot be processed until the access fault handler completes. Therefore, even for the most simple protocol event, the pure request/reply semantics are violated.

One advantage of this policy is that it easily scales. Since it uses a shared buffer pool, it does not require preallocation of buffers for each potential sender[1]. A second advantage is that it has minimal processing overheads since no credit counters need to be kept updated. Unfortunately, by itself this policy can lead to deadlocks with the generalized active messages supported by Tempest. In particular, if two nodes concurrently decide to send a message from within a handler when their receive queues are full and messages for each other wait in the overflow queue, no further progress can be made. Therefore, it is not surprising that the FM library did not guarantee deadlock-free behavior when new requests where generated inside message handlers.

Interestingly enough, this policy did not survive long in the FM library. In its next release [PKC97], it has been replaced by a standard credit-based flow control scheme. While no extensive details for the new policy have been published, it appears to be a standard credit-based protocol. By itself such protocol, it is not sufficient to support new requests generated inside message handlers. Since the FM library is not available in source form, no further comments are possible.

3.3.3 Sender-Overflow

The *sender-overflow* policy uses a credit-based flow control mechanism as its basis. When the credits are exhausted, the sender is not allowed to inject new messages in the network. Instead, it has to copy any such messages to an overflow queue in its local memory. Messages from the overflow queue will be injected in the network when new credits arrive. This policy has been employed in the Tempest libraries for the Typhoon parallel machines [RLW94] developed on the WWT-II simulator [MRF+97].

Sender-overflow was designed to deal with temporary buffer shortages. It is ideal for protocols with message activation graphs that each requires a small number of buffers and few of which are traversed concurrently. For such programs, it will not impose any extra overheads to the messaging operations since no buffering in the overflow queues occurs.

However, this policy fails to offer good performance for streaming protocols that require feedback from the messaging subsystem to limit the injection of new messages. When credits exhaust, the sender simply copies the data to the overflow queue. Since the sender is not throttled to match the receiver, it can flood the messaging subsystem, oblivious that its messages are copied to the overflow queue. Therefore, for the messaging subsystem to recover such messages blasts, it requires huge overflow queues that can easily exhaust all the swap space available on the sender. This situation can occur often because the cost of receiving a message is inherently higher than sending it due to the design of the Tempest messaging interface.

1. Credit-based flow control policies can dynamically adjust the number of credits assigned to each sender, and scale likewise. However, implementing such schemes is more complicated compared to the approach of the *return-to-sender* policy.

Moreover, architectural factors aggravate this problem. For example, on COW, loads from the NI memory are twice as slow as stores. Consequently, messaging patterns as simple as a one-way message blasts from one node to another can break the system.

The fundamental problem of this policy with such simple message patterns is that it is not a real flow control policy. A flow control policy must throttle the injection rate at the sender to the consumption rate at the receiver. Therefore, a proposed solution is to restrict the size of the sender-overflow queue [Woo96]. This implies that the number of messages that a handler can send to the same node is restricted likewise. Furthermore, no new handlers can be invoked if there are not enough buffers available for all potential message destinations.

While this approach addresses the issue of huge overflow queues, defining a limit on the number of messages that handlers are allowed to send is an arbitrary decision. It is difficult to propose a methodology for determining its optimal value or include it in a performance model presented to the user. Moreover, it will still fail to offer adequate performance for streaming protocols. Again, the fundamental problem is the assumption that buffer shortages are temporary and the peak in demand will soon level out. When this is not true, the policy will result in every message being streamed through the overflow buffer before it is injected in the network. Effectively, we waste processing power in the intermediate copy step with no performance benefit.

However, there are situations where disassociating the sender from the receiver through an intermediate buffer stage results in improved performance. For example, it is possible that one node is too slow to process messages. *Sender-overflow* allows the sender to proceed doing useful work until that node returns new credits. This property of “bandwidth matching” for multiple destinations allows the system to match the message injection rate to the message delivery rate across multiple destinations at the extra cost of a local copy. It can improve the performance of user protocols that exhibit unbalanced messages patterns as in this example.

3.3.4 Receiver-Overflow

Receiver-overflow breaks the buffer allocation problem into two subproblems: flow control and deadlock detection. This approach has been implemented in a number of systems including Blizzard. The Blizzard design has been optimized to eliminate adverse performance effects for those protocols whose buffer requirements do not exceed the available buffers.

This policy uses a standard credit-based flow control mechanism to guarantee that buffer space exists before sending a message. Unfortunately, this can cause deadlock if a user overcommits network resources, for example, if a message handler sends more messages than the low-level network layer can buffer. To prevent this type of deadlock, the system must extend the buffer space by copying messages to an overflow software queue in main memory [BCL+95,MKAK94] of the receiver.

Earlier systems aggressively buffer messages when the sender blocks [SFL+94, BCL+95] or after it has been blocked for a timeout interval [KA93]. However, the extra copies this entails can potentially degrade performance [MFHW96]. Blizzard, instead, uses a conservative deadlock detection scheme, which only buffers messages when a deadlock may have occurred. For this purpose, each node *n* uses a conservative, local condition to detect potential deadlocks while sending messages to node *i* from within a handler:

$$msgs\_sent\_to\_node(i)_n - acks\_received\_from\_node(i)_n \geq window\_size$$

and ∃ *node j* such that

$$msgs\_received\_from\_node(j)_n - acks\_sent\_to\_node(j)_n \geq window\_size$$

This condition is necessary but not sufficient for two processors to be blocked on message sends. If *i=j*, then a cycle of length two exists; if *i<>j*, then a larger cycle *may* exist. In either case, the NI copies messages from the low-level receive queue into a software overflow queue.

All but one of the counters that this condition requires are maintained by the host processor(s) in the user virtual memory. The only exception are the *acks_received_from_node* counters which are maintained by the NI as new credits arrive. This is required because otherwise the host would have to keep examining the messages pending in the receive queue to find and use any new credits that arrived after it has blocked.

Blizzard includes two further refinements in the basic algorithm. First, it batches and piggybacks acknowledgments. The system accumulates credits as messages are processed. If more than a fourth of *window_size* credits have been accumulated, an explicit acknowledgment is generated. Furthermore, explicit acknowledgments are silently consumed by the NI. Credits are piggybacked in outgoing messages destined for the node from which the processed messages arrived. Second, the implementation reserves one fourth of the window slots for messages generated from within protocol handlers. This is useful to ensure that the message handlers will not starve due to messages directly injected by the computation.

The proof for the deadlock condition is simple. If *n* nodes are deadlocked in a cycle then: ∀ $n: messages\_sent\_to\_node(n+1)_n - acks\_received\_from\_node(n+1)_n > window\ size$. However, ∀ $n: message\_sent\_to\_node(n+1)_n = messages\_received\_from\_node(n)_{n+1}$ and $acks\_received\_from\_node(n+1)_n = acks\_sent\_to\_node(n)_{n+1}$. Therefore, ∀ $n: messages\_received\_from\_node(n)_{n+1} = acks\_sent\_to\_node(n)_{n+1}$. So the condition is necessary. However, it not sufficient. Since each node has local knowledge, these counters can misrepresent the global state. For example, some acknowledgments might be on their way to one node but until it receives them it is not be aware of the global state. This can lead to unnecessary message copying to the overflow queue. In practice however, this rarely happens as it is evident from actual program statistics for four reasons. First, shared memory coherence protocols do not require more that the available buffers to begin with. Second, the deadlock condi-

tion is triggered only when messages are sent from the protocol handlers. Streaming protocols, which are used for application-specific coherence protocols, typically inject messages from the computation only. Third, when the message streams originate from within handlers and create a potential deadlock cycle that involve only two nodes, the piggybacked acknowledgments keep credits always available on the two nodes. Fourth, for the only remaining case, when the message streams originate from within handlers and create a potential deadlock cycle involve more than two nodes, the window of vulnerability is limited to about 3 μsecs. This how long it takes for an explicit acknowledgment to be injected through the network and consumed by the receiver's interface.

The designers of Berkeley active messages argued that targeting message-driven architectures was the wrong approach. Instead, the compiler itself should generate code to explicitly manage buffers in the message handlers while the message subsystem should be freed from this task. In this way, message handlers could exploit and optimize special cases [vECGS92]. The alternative approach followed by Blizzard[1] still allows the compiler to optimize special cases. However, it relies on a robust fallback mechanism to handle the general case. The advantage of this approach is that the system can use dynamic knowledge to handle exactly those cases that there was not enough static information to handle at compile time. This is more efficient than simulating a message-driven architecture with pure request/reply protocols given that the introduction of such mechanism does not cause adverse performance effects for simpler protocols. Moreover, as some researchers have pointed, targeting pure request/reply protocols does not always result in the most efficient use of the network [CCBS95].

3.3.5 Sender-Overflow Revisited

In Section 3.3.3, we show that *sender-overflow* is not a viable policy. However, it is the only one that exhibits the "bandwidth matching" property. Therefore, it is worthwhile to investigate whether it is possible to incorporate this policy in the more robust framework of the receiver-overflow policy.

Indeed, modest modifications are required to the receiver-overflow algorithms to accommodate sender-overflow. If we simply define the window size to be the sum of the available buff-

1. The focus of this discussion is limited to buffer allocation only. As in Berkeley active messages, Tempest does not support blocking handlers. Therefore, the compiler (or the user protocol for hand coded protocols) must explicitly manage handler continuations. Buffer allocation depends on the global system state while continuations depend on the local state. Therefore, it is easier to solve this problem without support from the messaging subsystem. Indeed, Teapot [CRL96], a high level protocol language and compiler developed for Tempest, supports blocking handlers. It hides the complexity of managing the handler continuations in the runtime library without adverse performance effects.

ers and the maximum sender-overflow queue, then the deadlock detection condition remains the same. Effectively, we can increase the window size beyond what the NI offers us.

While this modified policy allows to take advantage of “bandwidth matching”, it still suffers the same performance problems as the original *sender-overflow* policy. Again for streaming protocols, all messages will go through the intermediate sender-overflow buffer. Therefore, it depends on the user protocol whether such an optimization will improve performance. Moreover, to implement this policy in Blizzard, a notification mechanism is required to inform the host processor when new credits arrive. This can slowdown message processing. For these reasons, it has not been implemented in Blizzard.

3.4 COW Communication Infrastructure

COW nodes (Figure 3-5) are equipped with a Myrinet NI that connects them through Myrinet switches [BCF+95]. The NI is located on the I/O bus (SBUS [Mic91]), which supports processor coherent I/O transfers between the memory and I/O devices. The bus bridge supports an I/O memory management unit (IOMMU) which provides translation from SBUS addresses to physical addresses. The NI is controlled by a 5 MIPS 16-bit processor.

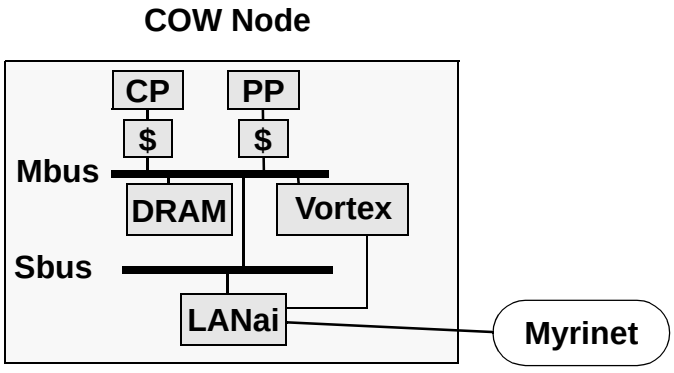

Figure 3-5. COW Hardware Organization.

With Myrinet hardware, the host (SPARC) processor and Myrinet LANai processor cooperate to send and receive data. The division of labor is flexible since the LANai processor is programmable. However, the SPARC processor is far faster, which effectively limits the LANai processor to simple tasks [PLC95]. Blizzard’s communication library started from the LANai Control Program (LCP) used in Berkeley’s LAM library [CLMY96]. The LCP was modified to fit the Tempest active message model and to improve performance for its expected usage. The changes are small, and, in fact, the modified LCP is still compatible with Berkeley’s LAM library. The protocol supports small messages up to 8 words and large messages up to 4 KB.

A Solaris device driver maps the Myrinet interface's local memory into a program's address space where it can accessed with uncached memory operations. To permit DMA between the LANai and the user address space, the driver allocates a kernel buffer mapped into the user address space.

The LCP implements, in LANai memory, separate send and receive queues. Each queue contains 256 entries consisting of a message header and up to 8 words of data. The low level packet format is presented in detail in Figure 3-6. The queues are fully addressable by the host processor since they are software structures in memory, implemented as circular buffers.

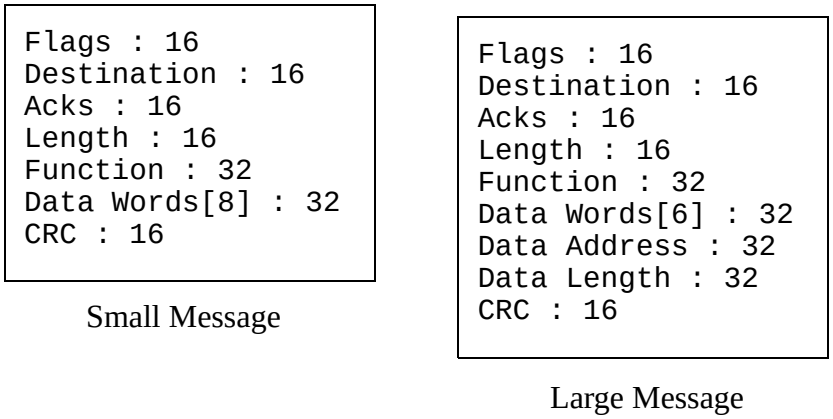

Figure 3-6. Low Level Data Format.

The host processor uses loads and stores to move the word data—the complete message for small messages and the header for large messages—in and out of LANai memory. The LANai processor uses DMA to move larger messages through intermediate kernel/user buffers. This organization (Figure 3-7) lowers the latency of small messages and increases the throughput of large messages. Extensive measurements verified that the queue organization is near optimal to achieve the lowest possible latency for small messages. While setting up the DMA transfers on the LANai introduces substantial overhead, DMA achieves higher bandwidth compared to uncached memory transfers. Therefore, it is the fastest way to move message data between the local memory and the LANai memory. Of course, the processor can move a small number of worlds faster but the cutoff point is surprisingly small. DMA is faster when moving more than 132 bytes from the main memory to LANai compared to uncached stores and more than 80 bytes from the LANAI to the main memory compared to uncached loads. The reason for the disparity between the two directions is that uncached loads have to bear the full latency twice one for the request and for the data. Uncached stores on the other hand are executed asynchronously through the processor's write buffer.

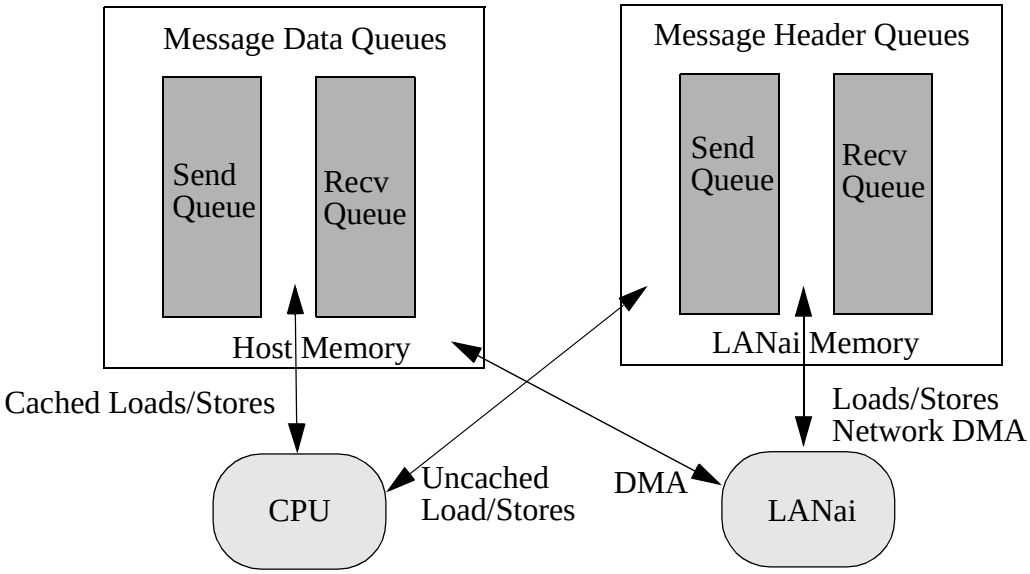

Figure 3-7. Message Queues And Accesses.

Directly accessing the LANai from the user process does not virtualize the NI, which limits network access to cooperating processes. However, it is possible to extend this functionality to multiple parallel programs by replicating the queue structures while using virtual memory hardware to enforce protection. An equivalent model is described in the last Active Message specification [MC95].

3.5 Transparent Message Notification

To reduce round-trip latencies, a node must quickly respond to arriving messages. Conceptually, an arriving message causes an interrupt, which invokes a message handler. In practice however, hardware interrupts are far too expensive and modern operating systems do not provide fast mechanisms to deliver asynchronous interrupts to the user process. For these reasons, active message implementations often resort to explicit polling primitives invoked directly by the application. Blizzard however, needs to support applications that are shared-memory codes, oblivious to the underlying communication. Therefore, we cannot expect them to explicitly poll. Instead, Blizzard supports implicit polling mechanisms. In this section, we will evaluate both interrupt-based and polling-based notification mechanisms that can be used for transparent message notification.

3.5.1 Interrupt Message Notification Alternatives

Blizzard/CM-5 [SFL+94], the first Blizzard implementation, used interrupts triggered by the NI as the message notification mechanism. This approach delivered the best performance in the CM-5 environment. A similar scheme has been also implemented for the Blizzard. However, it failed to offer good performance. In this section, first we will review the CM-5 and COW implementations. Then, we will examine why interrupts do not perform well on COW and I will finish by considering techniques to improve their performance.

In the CM-5, interrupts were delivered to user-level by the CM-5 node operating system efficiently (~300 cycles). To preserve atomicity, it was required that the user-level interrupts were disabled user-level while running in a handler. The CM-5 did not provide user-level access to the interrupt mask. Therefore, relative expensive kernel traps were required to both disable and re-enable interrupts. Instead, Blizzard/CM-5 used a software interrupt masking scheme similar to one proposed by Stodolsky, et al. [SCB93]. The key observation is that interrupts occur much less frequently than critical sections, so you should optimize for this common case. This approach used a software flag to mark critical sections. The lowest-level interrupt handler checked this “software-disable” flag. If it was set, the handler set a “deferred-interrupt” flag, disabled further user-level hardware interrupts, and returned. On exit from a critical section, the code first cleared the software-disable flag and then checked for deferred interrupts. After processing deferred interrupts, the user-level handler trapped back into the kernel to re-enable hardware interrupts. Stodolsky, et al.'s implementation uses a static variable to store the flags. To minimize overhead, Blizzard/CM-5’s scheme used a global register. A further complication was introduced because the system had to avoid livelocks and guarantee forward progress. Livelocks could occur as follows. After an access fault was resolved and the computation was restarted, if an interrupt was delivered before the application could execute the memory access that caused the fault, the block could be stolen. To avoid this scenario, after a fault interrupts were disabled and a timer was started. When the timer expired, interrupts were enabled again.

The COW implementation closely follows the Blizzard/CM-5 design. When a message arrives and the interrupts are enabled, the NI triggers an interrupt and disables further interrupts. Interrupts are disabled throughout the time that messages are being processed and whenever the user protocol waits for a message. Therefore, only messages that arrive asynchronously while user code is executing will pay the interrupt penalty. The interrupt is caught by the network device driver and a Unix signal is forwarded to the user process.The roundtrip time per notification is around 70 microseconds (~7600 cycles), nearly an order of magnitude higher than in the CM-5.

Published performance results [ZIS+97] from this implementation suggest that only in limited cases is this scheme competitive to other alternatives. In particular, it performs adequately when the program creates very little messaging activity or when it creates too much. In the latter case, as nodes compete for accesses to the same block, we observe a ping-pong effect as a block traverses from node to node. Interrupts can provide better performance because the

effect of false sharing is indirectly minimized since message processing is disabled and a timer is started that will deliver a signal to resume processing. The granularity of Unix timers is very coarse and so, interrupts allow more time for useful work before the block is taken away. Interrupts may also perform better because they increase the notification time and therefore again, there is more time for the node to do useful work before it gives up the block ownership to other nodes.

Since the overhead of the signal interface dominates the interrupt cost, it is worthwhile to investigate if it can be eliminated by techniques similar to the ones in the fast trap dispatch interface (Section 2.4.6) that speed up the delivery of synchronous traps.

The fundamental difference between synchronous and asynchronous traps (interrupts) is that when the former are caught by the kernel, it is guaranteed that they should be delivered to the thread that was just executing. This knowledge enables the fast trap dispatch interface fairly easily to change the program counter and redirect the computation to the fast trap dispatch handler without having to access complicated kernel data structures and interact with the processor scheduling algorithms. Interrupts however, can trigger a kernel exception at any moment without any regards to which process is currently executing on the processor that services the interrupt. Therefore, the biggest problem for fast interrupt is locating the process for which the interrupt is destined, especially on SMP machines.

The root cause of this problem is the fact that interrupts are considered synchronous to the thread that receives. They are expected to halt the process and invoke the handler. However, we can define the problem away by changing this abstraction to eliminate any synchronization guaranties for the handler invocation. In particular, it is possible to define a separate context to receive the interrupts. Then, using the same techniques as the fast trap interface, we can expect that the delivery of interrupt will an operation of the same order of magnitude as the delivery of synchronous traps using the fast trap dispatch interface. More specifically, we can have two cases.

- The interrupt is delivered to a processor that is currently executing a thread that has registered the notification. In this case, delivering the interrupt does not differ for delivering a synchronous trap.
- The interrupt is delivered to a processor that it is executing some other thread. Then, a context switch must occur to a context waiting to receive notifications. This will have higher overhead than the previous case. However, we still avoid the most important overhead component of asynchronous traps, locating and interrupting a specific thread.

While the scheme sounds attractive, it has not been implemented in Blizzard for two reasons. First, it requires some effort to implement the kernel code. Second, very early low-cost polling-based alternatives for transparent message notification were introduced in Blizzard.

3.5.2 Polling Message Notification Alternatives

Blizzard resorts to two forms of implicit polling. One approach relies on executable editing using the EEL tool [LS95] to insert explicit polling code on backwards loop edges and potentially recursive procedure calls. This code (listed in Figure 3-8) reads a location to test if a message arrived. If the conditions codes are not live, which is the common case, the inserted sequence takes seven instructions. Otherwise, it takes ten instructions. The second approach, on SMP nodes, uses one of the node's processors to poll and execute the message dispatch loop. In addition, this processor handles incoming traffic by running protocol handlers and sending replies. However, the scheme requires additional locking and it does result in the best utilization of this processor (Section 5.2). In either approach, polling can be implemented with one of four techniques: poll on the LANai receive queue, poll on a cachable memory location that the LANai updates by interrupting the kernel, poll on a cachable memory location that the LANai updates with a DMA transfer, or use the Vortex's hardware support for detecting message arrivals.

The original Berkeley library polls on the receive queue head, which requires an uncached memory access across the SBus-to-MBus bridge. These accesses are costly (0.38 μsecs) and polling consumes bandwidth on the MBus and SBus-to-MBus bridge.

The second and third alternative polls on a cacheable location in the kernel-user data buffer. This reduces the overhead of a "failed" poll to a cache hit and eliminates unnecessary memory bus traffic. Conversely, updating the flag is more expensive. We tried two approaches to update the flag when a message arrives. For the first, the LANai interrupts the host processor, which runs the device driver updates the flag. This requires 14 μsecs to update the flag. A better approach, requiring only 4 μsecs, is for the LANai to directly update the flag with a one-word DMA. However, it requires the NI to be able to modify a cacheable location in the user address space, which may not be possible for all NIs. The LANai keeps a local copy of the flag and only sets the host flag (or signals an interrupt to set the host flag) when a message arrives *and* its local copy is not already set. The host clears both its flag and the LANai copy once the message queue is empty. Thus, only the first message in a burst incurs the cost of an update.

The final alternative uses the message bit in Vortex's status register. The Vortex board is connected via a jumper to the Myrinet interface board's LED signal. The LANai toggles the LED by writing to a register upon message arrival, which in turn sets the Vortex message bit. Since the Vortex status register is cachable, "failed" polls are quick and do not cause bus traffic. However, a message arrival causes a cache miss followed by a handshake to clear the register. This approach requires 1.3 μsecs.

The polling technique is more important when the polls are inserted in the executable. In this case, polling is very frequent and more importantly delays the computation. The polling overhead depends on the frequency of backwards loop edges in the application. Moreover, it is very sensitive to compiler optimizations like loop unrolling that directly affect the fre-

```
ld [%g7 + poll_location], %g5    ! Load the poll location
ld [%g5], %g5                    ! Load the poll flag value
andcc %g5, 0x80, %g0             ! Check if set
be endlabel                      ! Check if set
sethi %hi(blz_handler), %g5      ! Call blizzard handler
jmpl %g5 + lo(blz_handler), %g6
nop
endlabel:
```

Fast Sequence (uses condition codes)

```
ld [%g7 + poll_location], %g5    ! Load the poll location
ld [%g5], %g5                    ! Load the poll flag value
and     %g5, 0x80, %g5           ! Check if set
srl     %g5, 0x5,  %g5
xor     %g5, 0x4, %g5
sethi jmplabel, %l0              ! set %l0 to inst at jmplabel
or      %l0, *1,  %l0
jmpl    %l0 + %g5, %g0           ! test if message pending
sethi %hi(blz_handler), %g5      ! Call blizzard handler
jmplabel: jmpl %g5 + lo(blz_handler), %g6
nop
```

Slow Sequence (does not use condition codes)

Figure 3-8. Poll Code Sequences.

The figure depicts the poll instruction sequences inserted in the application code using executable editing to implement implicit polling.

quency of these edges in the executable. Adding the polling code typically increases the execution time of the sequential program by up to 15% when we poll on a cacheable location.

However, it can climb up to 50% when we use an uncachable memory location in the LANai memory.

3.6 Blizzard Implementation Details

Blizzard's messaging subsystem has been extensively tuned to optimize the critical paths for the common case. Nevetheless, it is robust enough to operate with arbitrary complex protocols. In this section, we will discuss how its structure, allows it to exhibit these properties.

Blizzard's messaging subsystem is a fully multithreaded design that it is capable to push the communication architecture to its limits. The design relies on critical sections as its synchronization abstraction. Synchronization primitives are used to define the start and end of any lines of code that should be executed atomically with respect to all other lines of codes tagged with the same synchronization variable. Atomic sections are implemented in a different way depending on the platform and the system structure. For uniprocessors nodes, the primitives simply disable interrupts (in software as discussed in Figure 3.5.1) or polling for the duration of the critical section. For multiprocessor nodes with a dedicated coprocessor, the primitives are simply lock operations. For multiprocessor nodes without a dedicated coprocessor, the primitives combine the lock operations with the polling control.

Very few critical sections exist in the message processing code. They are small, typically a couple of lines of C code, which translate to a few tenths of machines instructions. This allows the messaging subsystem to scale to large SMP nodes. Its scalability has been verified using an experimental testbed, Blizzard/SYSV [1], which emulates high speed messaging using System V shared memory [Vah96]. Blizzard/SYSV runs on SMP workstations and servers and it has been used to tune the use of critical sections. For this purpose, a ES-6000 Sun Enterprise Server with twelve processors was used to emulate SMP multiprocessor nodes.

In COW, the performance of the messaging subsystem is limited by the communication hardware, since one processor is capable of fully saturating the I/O bridge. This limitation makes it difficult to identify bottlenecks in the design or implementation of the messaging subsystem. The Blizzard/SYSV experiments demonstrated that atomic handlers can also limit the performance on SMP nodes with more than two processors. Consequently, unlike Blizzard, Blizzard/SYSV supports non-atomic concurrent handlers. Locks are provided to support a restricted synchronization model as discussed in Section 3.1. Moreover, the Tempest interface needs to be extended to support the new synchronization model. The actual changes (Figure 3-9) are modest and they can completely hidden with appropriate C macros for back-

1. This is the first time that Blizzard/SMP is mentioned. It is based on Blizzard/S and it emulates the LANai message queues in System V shared memory. The design goal for its messaging subsystem was to remain as faithful as possible to the COW implementation with regards to how messages are scheduled and serviced.

```
Receive Primitives
handler function(src, message size, this message)

int TPPI_recv_W(this message)

void TPPI_recv_R(region address, this message)

void TPPI_recv_Ba(block address, this message)
```

Figure 3-9. Tempest extensions for Blizzard/SYSV.

The Tempest interface has been extended for Blizzard/SYSV, an experimental testbed that uses System V shared memory to emulate high speed networking.

```
Start atomic section (send queue)
If window is closed
          Check (and maybe recover) from deadlock
End
Get next entry in the send queue and advance queue pointer
End atomic section (send queue)
Start atomic section (receive queue)
Copy and clear pending acknowledgments
End atomic section (receive queue)
Copy data to send queue entry
Launch Message
```

Figure 3-10. Message Injection.

wards compatibility. However, Blizzard/SYSV's handlers are not atomic. Therefore, user protocols need to be modified to ensure safe accesses to their internal data structures.

3.6.1 Message Injection

Injecting a message into the network is a very simple operation (Figure 3-10). Sending a message requires synchronization with the message dispatch loop to safely access the counter of pending acknowledgments. The counter will be reset and its value piggybacked to the outgoing message. The most time consuming operation is actually transferring the data to the NI. Interestingly, this operation is not in a critical section.

3.6.2 Handler Dispatch

Compared to injecting a message, dispatching a message handler is quite complicated. The dispatch loop has to check for other protocol events such as access faults and launch the appropriate handlers. Moreover, there are different versions of the dispatch loop depending on whether the Blizzard implementation supports multiple processors and whether there exists a dedicated processor for message processing. In order to manage this complexity, we will first review the dispatch loop for Blizzard implementations with a dedicated network coprocessor, being the simplest ones, and then proceed to other cases.

Figure 3-11 lists the pseudo-code for the message dispatch loop with a dedicated network processor. The network processor repeatedly polls for access faults and messages and executes the respective handlers. It looks for messages in the receiver-overflow queue, in the local queue for messages send to itself or in the LANai queue. Messages in the receiver overflow queue have higher priority that any other messages so that messages are delivered in-order. While in-order delivery is not required by Tempest, experience has shown that it makes the development of user protocols easier. Dispatching the handler itself is a relatively simple operation. In a critical section, the queue pointer is advanced and the counter for pending acknowledgments is incremented. If too many acknowledgments have accumulated then an explicit acknowledgment is generated. Then, the handler is launched. The actual transfer of the message data will happen while the handler is executing when is calls the appropriate Tempest primitives (Figure 3-1). In Blizzard, the handler launch is done inside a critical section since Tempest dictates atomic handlers.

Figure 3-12 lists the pseudo-code for the message dispatch loop with executable editing. In this case, two versions of the message dispatch are required. The first is invoked when a message is detected from the polling code inserted in the executable. The second is invoked when the user protocol waits for a message. In this case, message processing must stop as soon as the waiting condition is satisfied to ensure forward progress and avoid livelocks.

A further optimization, not included in the presented pseudo-code is that the messaging subsystem attempts to avoid the notification cost when possible. For this reason, it does not use the polling location after at least one message arrives. Instead, it chooses to poll directly on the receive queue as long as new messages arrive back-to-back (or until the condition is satisfied for the second form of the dispatch loop with executable editing).

3.7 Performance

This section presents microbenchmarks to demonstrate the performance of the network. and establish that the Blizzard's extended functionality does not hinder it from being competitive with Berkeley active messages.

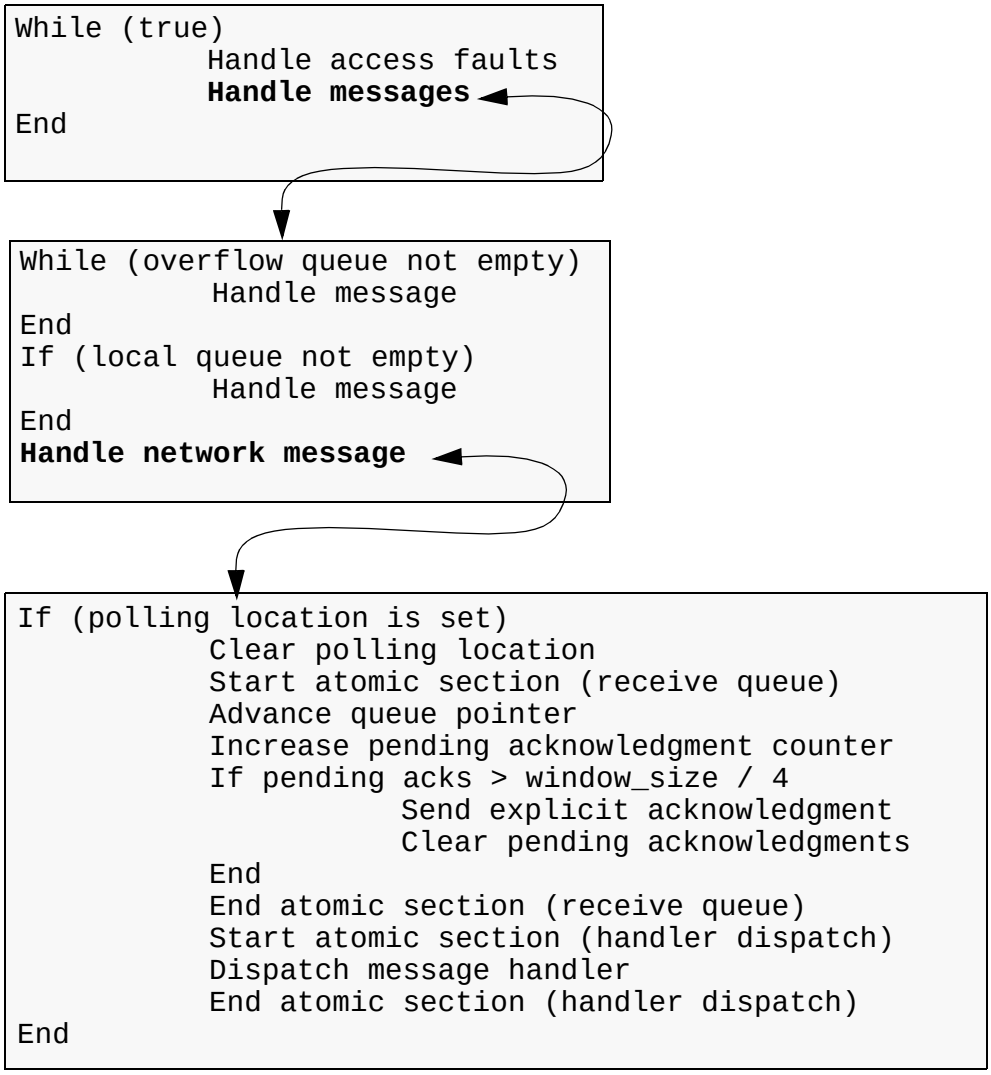

Figure 3-11. Message dispatch with a network coprocessor.

3.7.1 Small Message Latency

The small message benchmark measures the round-trip cost of small messages. In a timing loop, one COW node sends a small one-word message to another node, which responds with a similar message. Two versions of the this benchmark are used. In the first, the message injection is performed by the computation. In the second, the message injection is performed by the message handler that receives the reply to the previous request. Figure 3-13 presents the results for Blizzard systems with executable editing on uniprocessor nodes or a dedicated network processor. It also includes the results for the Berkeley active message library but only for the first version of the microbenchmarks since the protocol used in the second one is illegal for Berkeley active messages.

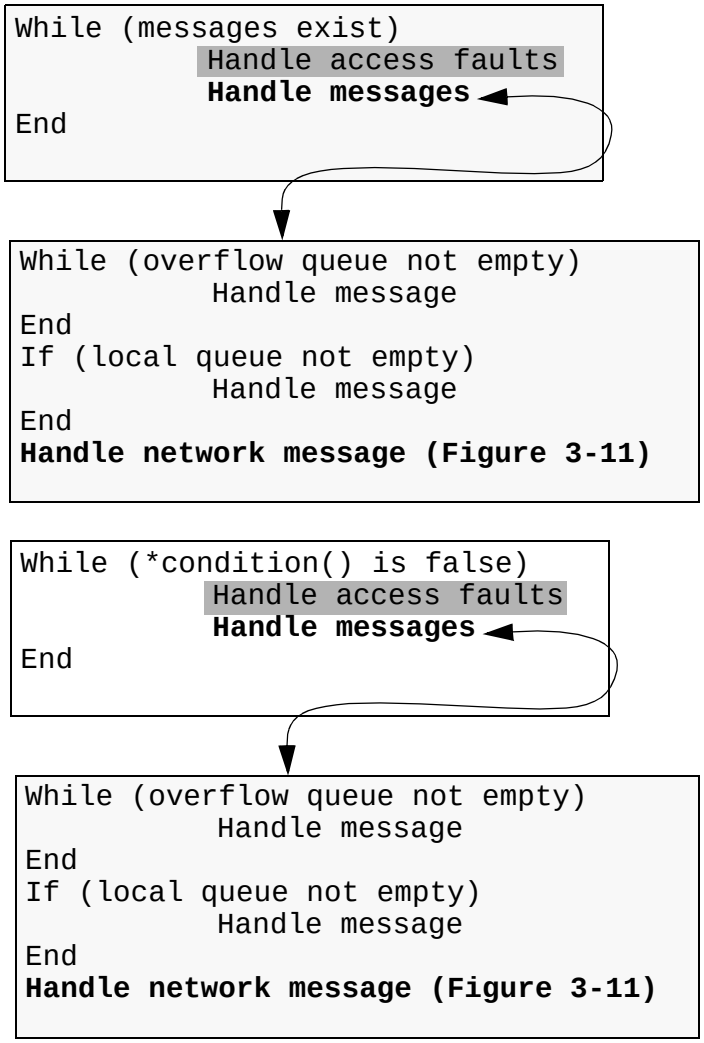

Figure 3-12. Message dispatch with executable editing.

With executable editing, different dispatch loops are used depending on whether the system entered the loop because the polling code detected a message or the user protocol expects a message. Note that grayed code is not present when the system is optimized for uniprocessor nodes.

The performance results clearly indicate that Blizzard's messaging subsystem is competitive with respect to Berkeley active messages (UCBAM). All polling techniques except setting the flag through interrupts perform very close to Berkeley active messages. To exclude the effect of the polling technique, we can compare the round-trip time in UCBAM with LANAI memory polling and executable editing under Blizzard. UCBAM messages achieve a roundtrip time of 38.7 μsecs vs. 40.1 μsecs when the message is injected by the computation and 35.3

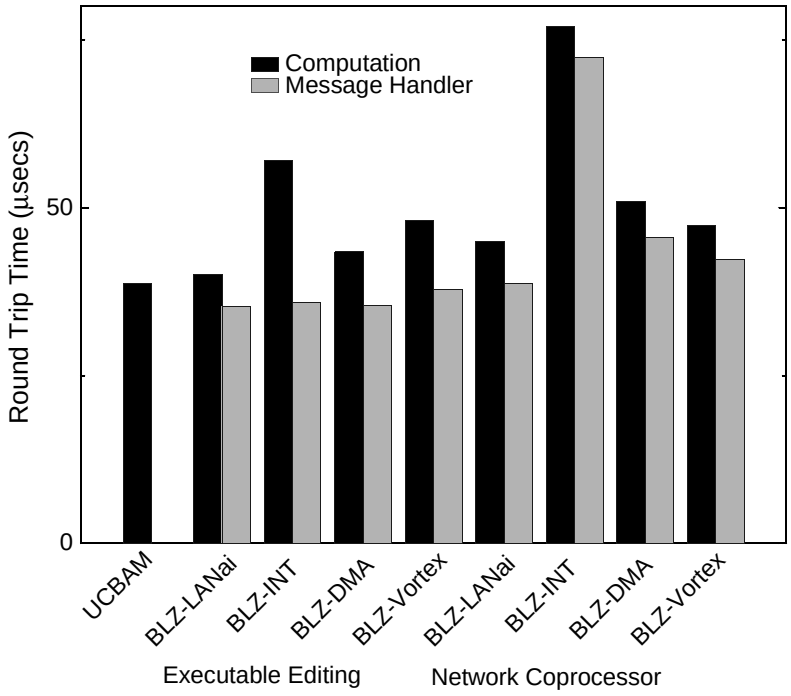

Figure 3-13. Small Message Latency.

The figure presents the round-trip time for one-word small messages. Blizzard polls for incoming messages in one of four ways: on a location set by the LANai by a DMA transfers ("DMA"), on a location set by a processor interrupted by the LANai ("INT"), on an uncachable location on in the LANai's memory ("LANai"), or a cachable control register provided by the Vortex board ("Vortex"). It uses two polling methods: "Executable editing" which inserts the polling code in the executable and "Network Coprocessor" which uses a second SPARC processor for message processing. For comparative purposes, the round-trip time using the Berkeley active messages library are included ("UCBAM").

µsecs when the message is injected by the message handler. Therefore, the Blizzard message subsystem, despite its general functionality and its ability to handle more general messaging patterns, it remains competitive performance-wise to UCBAM messages.

Comparing the polling techniques, we see polling on the LANai memory achieves minimum latency. However, we should not forget that failed polls consume bus bandwidth even if it does not affect performance in this microbenchmark.

With executable editing, when messages are injected from the handler polling is performed always on the receive queue directly. Therefore, the performance is very close to the LANai memory polling. In fact, the only reason that it is slightly slower is because the polling flag still has to be cleared. For this reason, the Vortex technique, which requires two uncached stores and two Mbus transactions to invalidate the message bit and read a fresh copy of the control register, is the slowest. In contrast, when the flag is set with DMA or interrupts, the system only has to clear a cacheable memory location. When messages are injected from the computation, not only the system has to return control to the application code, but it also incurs the notification cost on the sender since it exited the message dispatch loop. Therefore, for all methods performance deteriorates. Moreover, we start seeing the differences that the polling techniques introduce to the notification cost.

With a dedicated network processor, the results are simpler to interpret. Notifying the computation about the message arrival incurs extra penalty but it is constant for all polling techniques. Moreover, the system incurs the notification cost both on the sender and the receiver and the difference among the polling techniques becomes more prominent.

3.7.2 Large Message Latency

The large message benchmark measures the round-trip time for a large message. The data block varies from 32–4K bytes (the maximum size supported). In all other respects, it exactly similar to the small message benchmark. This time only the results from LANai memory polling and DMA polling are presented for Blizzard (Figure 3-14). The performance effect of the polling techniques is similar as for small messages. DMA polling is included because it directly corresponds to UCBAM messages (i.e., both poll on LANai memory). Similarly, the DMA polling is included because it performs best with executable editing (i.e., in uniprocessor systems) from all the polling techniques that use a cacheable location.

As for small messages, Blizzard is competitive with UCBAM messages. Blizzard performs worse for small data size but it catches up as the message size increases. Again, we can compare the round-trip time in UCBAM with LANAI memory polling and executable editing under Blizzard to exclude the effect of the polling technique. For 32 byte messages, UCBAM messages achieve roundtrip times of 58.9 μsecs vs. 67.1 μsecs when the message is injected by the computation and 63.1 μsecs when the message is injected by the message handler with LANai polling. For 4Kbyte messages, UCBAM messages achieve roundtrip times of 855.7 μsecs vs. 853.1 μsecs when the message is injected by the computation and 848.9 μsecs when the message is injected by the message handler with LANai polling. Blizzard performs better with larger messages due to its optimized (for blocks aligned at a 32-byte boundary) memory copy routine.

For small data block sizes, the performance effect of the polling technique is exactly the same (numerically) as for small messages. As the data block size increases, the LANai method starts to interfere with the data block transfers between the LANai and the host memory. This becomes more evident with a dedicated network processor, when messages are

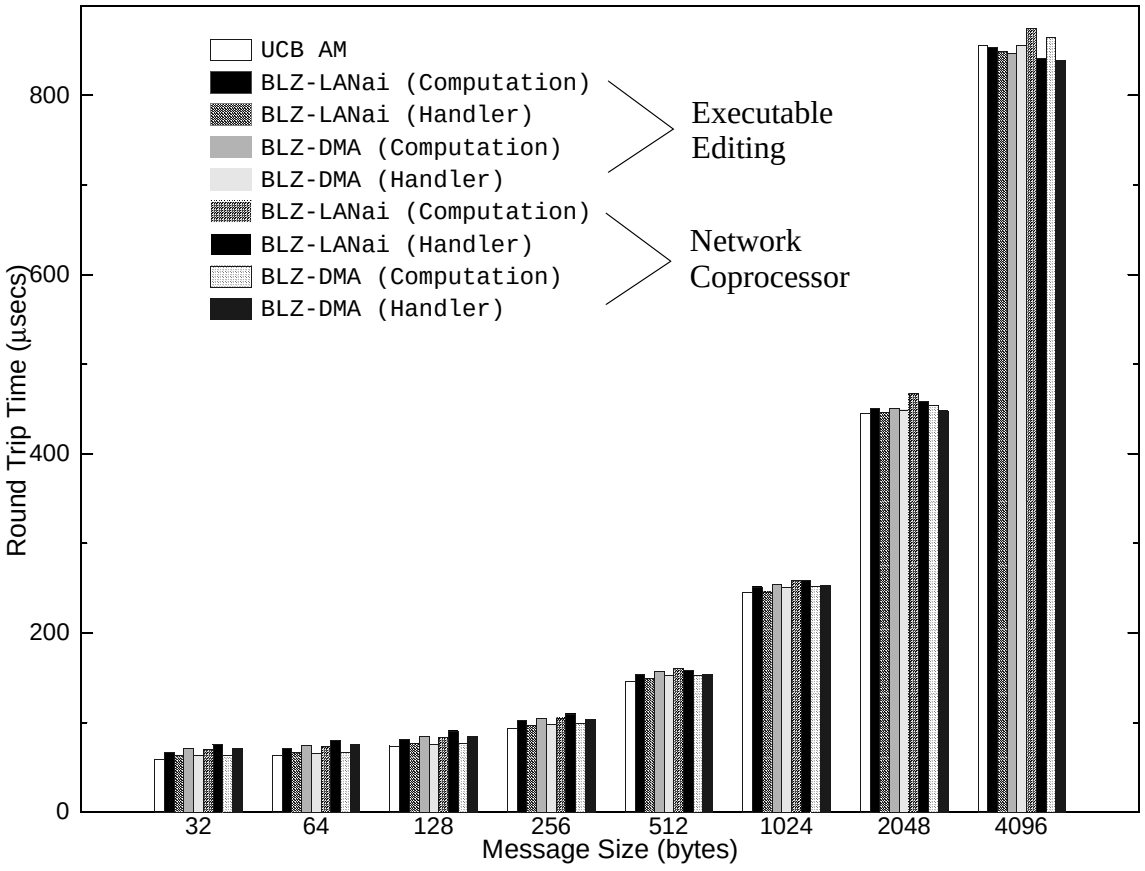

Figure 3-14. Large Message Latency.

The figure presents the round-trip time for large messages of varying size. Blizzard polls for incoming messages in one of two ways: on a location set by the LANai by a DMA transfers ("DMA") and on an uncachable location on in the LANai's memory ("LANai"). It uses two polling methods: "Executable editing" which inserts the polling code in the executable and "Network Coprocessor" which uses a second SPARC processor for message processing. For comparative purposes, the round-trip time using the Berkeley active messages library are included ("UCBAM").

injected by the computation. Continuous polling on LANai memory by the network processor slows down the DMA to and from the LANai.

3.7.3 Large Message Throughput

The throughput benchmark measures the maximum throughput through the network layer for large messages by blasting messages from one node to another as fast as possible. Overall, 64K messages are being send. For 4 Kbytes messages, the data stream is 512 MBytes long. The results from this experiment are presented in Figure 3-15 for the same systems as for the latency experiment.

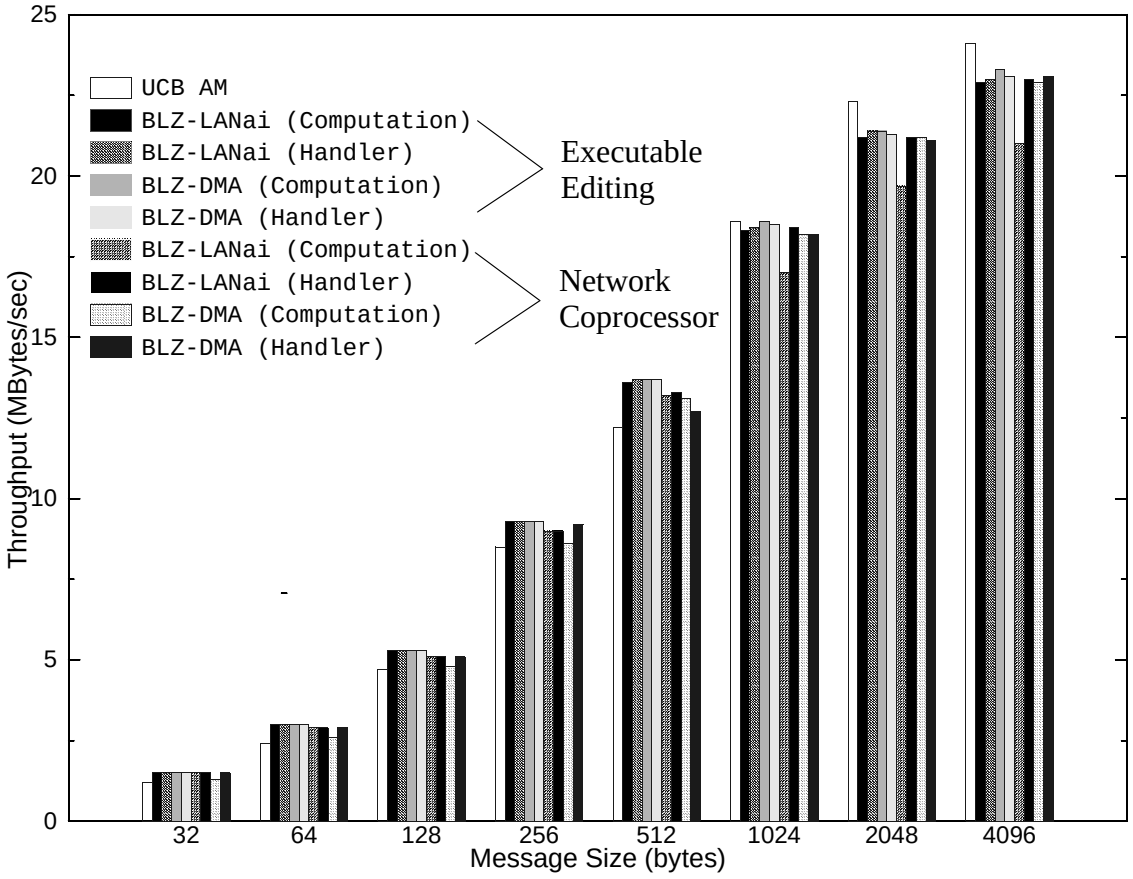

Figure 3-15. Large Message Bandwidth.

The figure presents the round-trip time for large messages of varying size for the same systems as in Figure 3-14.

Once again, the performance results validate my assertion that Blizzard is competitive with Berkeley active messages. Blizzard achieves better performance for small messages and slightly worse performance for larger messages. One of the reasons that explains this result is that the UCBAM library enforces request-reply semantics, but allows for requests without replies. In that case, the receiver automatically sends a null reply. In one-way transfers, this

approach generates a null reply—an explicit acknowledgment—for every message. By contrast, Blizzard's buffering policy sends one reply for every fourth of *window_size* messages (one explicit acknowledgment for every four messages). Since the relative cost of processing the acknowledgments is higher for smaller messages, Blizzard performs better. UCBAM messages are optimized towards bandwidth which has been a secondary consideration in Blizzard. The maximum achieved value of 23-24 MB/secs with 4Kbyte messages for all systems is close to reported Myrinet measurements [CLMY96,PLC95].

Finally, this microbenchmark provides a good example that LANai memory polling can affect performance even if it is the fastest notification mechanism. In the case of a dedicated processor, when the messages are injected by the computation, polling on LANai memory consumes scarce bus bandwidth, while a new message is injected by the computation processor.

3.8 Conclusions

This chapter focuses on Blizzard's messaging subsystem. Blizzard supports the Tempest messaging interface that it has specifically designed to support user level shared memory coherence protocols. Tempest's messaging interface is influenced by Berkeley active messages. However, due to its intended use, Tempest provides more general functionality than Berkeley active messages: it supports protocols without pure request/reply semantics and it does not rely on explicit polling from the application to receive incoming messages. This generality makes the implementation of Tempest a challenging proposition. In this chapter, I approach the challenge in a methodical way. Therefore, the solutions apply beyond the Tempest world and constitute a practical, effective way to generalize the Berkeley active message model.

First, I address the issue of user protocols not limited to request/reply semantics. I presented a methodology to determine an upper limit to the buffer requirements of user protocols. Then, based on this methodology, I classify popular parallel programming models according to the buffer requirements of the protocols that generate. Incidentally, the analysis reveals that fine-grain shared memory coherence protocols, have limited buffer requirements. Subsequently, I evaluate different buffering policies and discuss their strengths and weaknesses. The discussion includes Blizzard's buffering policy, which is based on receiver-overflow queues. Unlike previous policies based on the same approach, it uses partial knowledge about the system state to buffer messages only when it is absolutely necessary to break deadlocks.

Second, I address the issue of message notification without explicit polling. I argue that the cost of delivering hardware interrupts is prohibitive in modern operating systems and I propose a scheme that can address this issue. Then, I focus on the current Blizzard solutions that relies either on polling code inserted by the application or a separate dedicated network processor.

Subsequently, I change the focus of the discussion to the internal details of Blizzard's messaging subsystem. It is a robust, fully multithreaded, scalable and versatile design. The same code is used on both uniprocessor and SMP nodes. It has been carefully tuned to optimize the critical paths. Finally, this chapter finishes with microbenchmarks that established that despite the general functionality, the performance of Blizzard's messaging subsystem remains competitive with the Berkeley active messages.

Chapter 4

Blizzard Performance

This chapter evaluates Blizzard as a platform for parallel applications. Fine-grain distributed shared memory (FGDSM) systems support shared memory by controlling memory accesses in a fine-grain manner. They trap on invalid memory references. Then they execute a shared memory coherence protocol in software. The protocol exchanges messages that transfer data and manipulate the fine-grain tags to support the shared memory semantics. Therefore, the performance of FGDSM systems depends both on the overheads associated with the fine-grain access control and messaging mechanisms. So far we have discussed the implementation and performance of the different mechanisms in isolation. In Chapter 2, we covered the different fine-grain tag implementations supported by Blizzard. In Chapter 3, we covered in detail Blizzard's messaging subsystem. We are now able to put all the pieces together and examine the complete picture.

FGDSM systems [SFL+94,SFH+96,SGT96] on networks of workstations carve a middle path between hardware and page-based software shared memory approaches. On one hand, their performance characteristics are worse than hardware shared memory [LL97], albeit with little or no hardware support and presumably at a lower cost point. On the other hand, unlike modern software page-based shared memory approaches [KDCZ93], they can support shared memory without relaxing the consistency model from sequential consistency [KDCZ93], while still offering the same or better performance [ZIS+97]. Unique among existing FGDSM systems is Blizzard's ability to eliminate the cost of shared memory and offer to the application the raw messaging performance. Blizzard supports the Tempest interface [Rei94] which exposes to user level the access control and messaging mechanisms. Thereby, Blizzard allows the development of application-specific coherence protocols [FLR+94] that optimize the data transfers to match the application sharing patterns. Application-specific protocols are particularly appropriate for FGDSM systems on low-cost platforms because their relative performance benefit is significantly higher than other proposed tightly-integrated high-end Tempest implementations [RPW96].

This chapter examines whether FGDSM systems on networks of workstations are a realistic platform for parallel applications using both low level measurements and application benchmarks. The question is answered with a qualified “yes”. Parallel applications with small communication requirements easily achieve significant speedups using transparent shared memory. Parallel application with high communication requirements require application-specific coherence protocols, selectively employed on the performance-critical data structures.

The rest of the chapter is organized as follows. Section 4.1 presents low-level performance details of the Blizzard implementations on the COW platform. Section 4.2 presents the parallel application suite that is used in further evaluations. Section 4.3 examines the application performance and how different application interact with the FGDSM system. Section 4.4 evaluates Blizzard as a platform for parallel programs. Finally, Section 4.5 presents the conclusions of this study.

4.1 Low Level Performance Characteristics

Parallel applications on shared memory platforms access local and remote memory. Therefore, application performance depends on a platform’s performance characteristics with regards to local and remote accesses. In this section, I present microbenchmark performance results that measure the latency and bandwidth of local and remote memory accesses. The performance characteristics of the local and remote memory references are measured using microbenchmarks [SGC93], simple small programs that measure the performance characteristics of specific platform features. The experiments compare and contrast the memory performance of remote and local memory references.

Sequential programs access the local memory when they miss in the processor cache. These misses are serviced by the processor cache controller. On COW, the controller brings 32 bytes into the processor cache both for read and write misses since the HyperSparc processor is running with the cache in writeback mode [ROS93].

For parallel programs on FGDSM systems, the situation is more complicated. As for sequential programs, they can cause local misses when accessing local data. They can also cause local misses when accessing remote data for which a copy exists locally. However, when they access remote memory and do not have a local copy or permission to perform the attempted operation on the local copy (e.g., write to a readonly block), then a protocol event is triggered and the FGDSM system must bring the remote data for other nodes. In these experiments, the FGDSM system transfers data in 128-byte chunks since most applications show the best performance with an 128-byte stache block size.

The first microbenchmark measures the remote memory latency. One node allocates huge array locally. Then, a second node references one word out of every stache block of the array in a tight loop. The total time for the loop is averaged over the number of blocks touched. To avoid including the page fault overhead, the node first touches a single block out of every array page before it proceeds to execute the timed loop. Table 4.1 presents the latency times

for common protocol events (read miss on invalid, write miss on invalid and write miss on readonly). The results are in agreement with what we would expect based on the access and messaging overheads for these events. With the exception of the fine-tag implementations (Blizzard/E, Blizzard/ES) that require a system call in some cases to modify the access control state, the event times are dominated by the messaging cost (90% of the total time). Blizzard/S shows the best performance (51-80 μsecs) due to low cost of fine-grain access control. Blizzard/E shows the worst performance (85-143 μsecs) due to the system call overhead on the critical path to set invalid ECC. In contrast to these results, local references that miss in the processor cache take only 0.3 μsecs. Therefore, remote references are 200-500 times slower than local ones. It immediately becomes obvious that accessing remote memory is a very expensive proposition.

Table 4.1: Remote memory latency.

The table lists the miss latency times for different access and miss types. "Read on Invalid" and "Write on Invalid" measure the latency for reads and writes to an invalid stache block. "Write on Readonly" measures the latency for writes to read-only shache blocks. Contrast these numbers with the local memory latency of 0.3 μsecs for a 32 byte cache block on the COW workstations.

Access Latency (μsecs)	Read On Invalid	Write On Invalid	Write On Readonly
Blizzard/T	88	90	59
Blizzard/E	85	143	71
Blizzard/S	80	79	51
Blizzard/ES	80	134	67

The second microbenchmark is similar to the first one but it measures the memory bandwidth. This time the second processor issues doubleword loads or stores that reference every location of the array in a tight loop. In every loop iteration, four doublewords are accessed. Table 4.2 presents the memory bandwidth results for loads or stores to both remote memory and local memory. Surprisingly, this microbenchmark reveals all the factors related with the fine-grain tag implementation that affect memory performance. The results for all tag implementations agree with the latency results with the exception of Blizzard/E. When writing remote memory with Blizzard/E, the remote memory bandwidth is reduced for two reasons:

the page protection changes from read-only to read-write once for every page and almost every store is executed within the kernel since until the last block has been touched the page protection is set to readonly.

The microbenchmark also reveals the effect of the FGDSM system on local memory bandwidth. For all fine-grain tag implementations, it is affected by the polling overhead. In every iteration of the inner loop, a poll on a cacheable memory location is performed to check for incoming messages. Moreover, for software tag implementations, the local bandwidth is further reduced due to the software access checks inserted in the application code. As a result of all these overheads, the local bandwidth can be reduced by as much as 50% compared to the maximum attainable values of 66 and 55 MB/sec for loads and stores respectively on the COW workstations when no FGDSM overheads are present. Again, transferring data through the FGDSM memory proves to be very inefficient although the difference is not as dramatic as for memory latencies. Blizzard/T and Blizzard/S provide the best remote bandwidth which is 25-50 lower than the maximum attainable local bandwidth without FGDSM overheads.

Table 4.2: Remote and local memory bandwidth.

Remote memory references trap to the Blizzard system once for every stache block (128 bytes). "Remote Read" and "Remote Write" list the access bandwidth for remote read and write references to invalid blocks. "Remote Upgrade" lists the access bandwidth for write references to readonly blocks. Local memory references ("Local Read", "Local Write") never trap. However, software tag implementations include the code inserted to check the validity of the reference. Moreover, for all implementations, the inner loop polls the network once for every 32 bytes accessed. Without these overheads, the maximum attainable values on the COW workstations are 66 and 53 MB/sec for reads and writes respectively.

Memory Bandwidth (MB/sec)	Remote Read	Remote Write	Remote Upgrade	Local Read	Local Write
Blizzard/T	1.4	1.4	2.1	55	43
Blizzard/E	1.3	0.8	0.8	55	43
Blizzard/S	1.5	1.5	2.1	44	25
Blizzard/ES	1.5	0.9	2.0	55	25

The remote bandwidth results show that FGDSM memory is also inefficient compared to messaging. In Chapter 3, we saw that the message throughput with 128-byte messages is 5 MB/sec, which is three times higher than what FGDSM shared memory achieves. Moreover, the message throughput with 4-Kbyte messages is over 20 MB/sec, or about one third of the maximum attainable local bandwidth. Fortunately, Blizzard supports the Tempest interface that exposes the access control and messaging mechanisms. Therefore, it can harness and offer the raw messaging bandwidth to the application, as we shall further discuss in Section 4.4.

4.2 Parallel Applications

The application set used to evaluate the overall performance of Blizzard consists of nine scientific applications and kernels. They were originally written for sequentially consistent (transparent) hardware shared memory systems. Sequentially consistent page-based systems perform poorly on most of these codes because of their fine-grain communication and write sharing [CDK+94, SFL+94]. Page-granularity systems with weaker consistency models however, offer competitive performance to sequentially consistent software FGDSM systems [ZIS+97] but complicate the programming model. Besides the nine transparent shared memory (TSM) versions, for three of the applications custom protocol (CS) versions have been developed by the WWT research group [FLR+94,MSH+95]. The custom protocols attempt to eliminate the access control overhead and directly harness the messaging performance of the COW platform by optimizing the data transfers to match the application sharing patterns.

In the rest of this section, I review the applications in detail. For each one, I mention the scientific problem that it solves. Then, I examine how it is parallelized. In particular, I focus in its data structures, the data partitioning among the processors and its sharing patterns.

Appbt is a NAS Parallel Benchmark [BBLS91] produced by NASA as an exemplar of computation and communication pattern in three-dimensional computational fluid dynamics applications. At each time step, it performs three computation phases in each of three dimensions. In the first phase, it calculates a block tridiagonal matrix *A*. In the second phase, it solves $Ax=b$ for x using Gaussian elimination. In the third phase, it recomputes the right hand side vector b. To parallelize the application, the matrix is partitioned into subcubes, each assigned to a different processor. On sixteen processors, the partition algorithm creates 4x2x2 subcubes. Sharing occurs in the second phase along the faces of the subcubes, where along the z dimension the processors are arranged in a pipeline order.

We examine both TSM and CS versions. Their main difference is how they enforce the pipeline order and how they transfer remote data. In the TSM version processors spin-wait on a shared array of counters. An element in the array corresponds to an 8x8x8 subcell in the matrix. Each processor p along the pipeline waits for the counter to become equal to p, perform the computation along the dimension on its data and then it updates the counter to $p+1$. The CS version addresses the synchronization problem by transferring data with messages which implicitly synchronize the processors in the correct order.

Barnes is one of the SPLASH-2 benchmarks [SWG92]. It simulates the evolution of bodies in a gravitational system. Its primary data structure is a shared oct-tree, which represents bodies' locations in space. In each iteration, the tree is constructed from scratch. Then, the forces exerted on each body are computed by using the oct-tree to find near-by bodies and approximate distant bodies. The bodies' location and speed are updated. The whole process is repeated in the next iteration. To parallelize the application, the processors cooperatively build the tree at the start of each iteration. Then, the bodies are partitioned among the processors so that each processor owns bodies that are closely located in the physical space to each other and the total work is equally divided among the processors. Finally, each processor computes the forces exerted on its subset of bodies. The tree build phase is responsible for the majority of the communication. Accesses during this phase do not exhibit any regularity and moreover, the initial data placement is random. Therefore, the application does not exhibit any regular sharing patterns.

EM3D models the propagation of electromagnetic waves through objects in three dimensions [CDG+**93**]. The problem is formulated as a computation on a bipartite graph with directed edges from *E* nodes, which represent electric fields, to *H* nodes, which represent magnetic fields, and vice versa. The main loop computes the change in *E* and *H* values over time. In each iteration, new *E* values are computed from the weighted sum of neighboring *H* nodes, and then new *H* values are computed from the weighted sum of neighboring *E* nodes. To parallelize the application, the bipartite graph is partitioned among the processors. No locking is required to update the graph nodes since the two phases are separated with a barrier. The program accepts as input the characteristics of a graph. The graph is described by the number of graph nodes N_G, the degree of each graph node *E*, the percentage *P* of remote edges (edges that link graph nodes that belong to different processors), and the maximum distance *D* between linked graph nodes in terms of processors.

We examine both TSM and CS versions of this application. In the TSM version, the processors transparently access remote data as they compute the new values of their graph nodes. Therefore, the computation is interleaved with communication. The CS version [FLR+**94**] separates the computation and communication in distinct phases. Instead of the moving the data through shared memory, the node that produced the new values is responsible for pushing them to the nodes that shall need them in the next iteration. Data are batched in large (4-Kbyte) messages, which takes advantage of the superior bandwidth achieved with large messages.

LU is one of SPLASH-2 kernels [SWG92]. It performs the blocked LU factorization of a dense matrix. This version partitions the matrix into blocks and the data for each block are allocated sequential in memory. There are three steps to the algorithm. First, a diagonal block is factored. Second, the current row and column blocks are updated. Third, the subsequent column blocks are updated using the factored block. The process is repeated until the matrix has been factored. The processors are positioned along a rectangular grid and the grid is repeated for all the blocks in the matrix.

Ocean is based in the non-contiguous version of the SPLASH-2 benchmark [SWG92]. It simulates eddy currents in an ocean basin. In the original SPLASH-2 application, processors shared data with their neighbors in a two-dimensional partition of the problem. In this version however, the problem grid is partitioned rowwise among the processors. This partition results in an increase in the overall data shared among processors, which depends in the perimeter to area ratio. However, at the same time it decreases fragmentation and therefore, the amount of useless data transferred (when sharing along the column a whole block is transferred to access a single word). Therefore, it results in better overall performance, especially on FGDSM systems such as Blizzard [ZIS+97].

Moldyn is a molecular dynamics application [BBO+83]. Molecules are uniformly distributed in a cubical region and the system computes a molecule's velocity and the force exerted by other molecules. An interaction list (rebuilt every 20 iterations) limits interaction with molecules within a cut-off radius.

We examine both DSM and CS versions. The TSM version makes effective use of shared memory [MSH+95]. The program's reduction phase performs operations on large arrays with minimum locking. However, in this phase, each processor accesses a large portion of the shared memory, which limits Blizzard's speedups. The CS version used explicit messages to avoid cache misses in the reduction phase. Moreover, once the fine-grain communication of the reduction phase has been eliminated, the program can use a larger stache block size (1024 Kbytes) to reduce the access control overhead.

Tomcatv is a parallel version of the SPEC benchmark [SPE90]. The input matrix is partitioned among the processors in contiguous rows. Sharing occurs across the boundary rows (similar to *ocean*) and therefore, it exhibits nearest neighbor communication.

Waterns (n squared) is one of the SPLASH-2 applications. It simulates a system of water molecules in liquid state using a brute force O(n2) algorithm with a cutoff radius. The main data structure is an array of water molecules allocated contiguously in memory. To parallelize the application, the molecule array is partitioned among the processors with each one assigned a contiguous portion of it. Each processor updates its own molecules. Then, it proceeds to update then molecules that belong to other processors. Locks are used to synchronize accesses from different processors.

Watersp is the spatial version of the SPLASH-2 benchmark. It solves the same problem as *waterns*. However, it uses different data structures and algorithms. The physical space is broken into cells. Each cell is assigned the list of molecules that are physically present in the cell. For each molecule, only the interactions with molecules in the same or neighboring cells have to be considered. To parallelize the application, each processor is assigned a contiguous cubical partition of cells together with the list of molecules assigned to those cells. For 16 processors, the partition algorithm creates 2x2x4 subcubes and sharing occurs among nodes that their subcubes touch at a base, edge or point.

4.3 Application Performance

We now proceed to examine the performance of parallel applications. In the results presented in this section, message notification is done with the polling code inserted directly into the executable as discussed in Chapter 3. A dedicated network processor is not used because it does result in the best utilization for that processor [FW97]. Chapter 5 discusses alternative ways to employ more processors per node in FGDSM systems.

The inserted code polls on a cacheable memory location. On message arrivals, this memory location is updated by the network interface using DMA. This is the preferred technique when the polling code is inserted into the executable since it gives the best performance (Section 3.5). Table 4.3 lists the applications, their input parameters and the execution times of the sequential versions (without FGDSM overhead) on a single COW node.

Table 4.4 presents the application speedups. As it is expected, the speedups vary depending on an application's inherent parallelism, and its interaction with the FGDSM system. Overall, the results demonstrate that Blizzard/T, which provides support for fine-grain sharing in hardware, always achieves the best speedups. The TSM programs show good speedups (above eight) in five applications, medium (between five and eight) in three applications and poor (below five) speedup in one application. The CS versions show good speedups in two applications and almost good speedup in one. Blizzard/E's speedups range from good (close to Blizzard/T) in some applications to very bad in the rest. Blizzard/S always incurs instrumentation overhead and therefore, performs worse than Blizzard/T. Unlike Blizzard/E, it follows the same performance trends as Blizzard/T at a lower performance level. Blizzard/ES either outperforms both Blizzard/E and Blizzard/S or performs close to the best of the two.

While the application speedups provide the overall picture, they are not sufficient to demonstrate the interactions between different fine-grain tag implementations and the application performance. For this purpose, I present performance results from application runs with timing code inserted in the Blizzard system to measure where the time is being spent.

Timing real systems as opposed to simulated ones may lead to misleading results due to perturbation introduced by the timing code. However, I argue that it does not appear to be a problem in these experiments for three reasons. First, extra care was taken to use low cost timing functions (5 μsecs per call), very sparingly in few critical entry points into the system. Second, these applications are static codes that do not change their access patterns depending on the duration of protocol events. Third, the performance effect of the timing functions is not very prominent (the parallel execution times are increased by less than 2%). These arguments suggest that the timing functions does not perturb the results to make them unusable.

In the rest of this section, first I concentrate on the performance of the TSM versions. Then, I discuss the implications of custom protocols as the performance of the TSM and CS versions is contrasted. Finally, I close the section with low-level statistics collected by the stache proto-

Table 4.3: Applications and input parameters.

The application set consists of nine shared memory scientific codes. The table lists the applications, the scientific problem they solve, the input data set and the time it takes (in secs) to run the sequential application (without FGDSM overhead) on a single COW node.

Application	Problem	Input Data Set	Sequential Time (secs)
appbt	3D implementation of CFD [BM95]	40x40x40 matrices, 4 iters	110
barnes	Barnes-Hut N-body simulation [WOT+95]	16K particles, 4 iters	33
em3d	3D electromagnetic wave propagation [CDG+93]	128K nodes, degree 6, 40% remote edges, distance span 1, 100 iterations	166
lu	Contiguous blocked dense LU factorization [WOT+95]	512x512 matrix, 16x16 blocks	75
ocean	Simulation of eddy currents [WOT+95]	514x514 “ocean” size	62
moldyn	Molecular dynamics [BBO+83]	8722 molecules, 40 iterations.	128
tomcatv	Thompson’s solver mesh generation [SPE90]	512x512 matrices, 100 iters	147
waterns	Water molecule force simulation [WOT+95]	4096 molecules	273
watersp	Spatial water molecule force simulation [WOT+95]	2744 molecules	65

Table 4.4: Base system speedups for sixteen uniprocessor nodes

Application speedups running on sixteen nodes. The speedups are measured with respect to an optimized sequential (no FGDSM overhead) implementation of the applications running on one node of the machine (see Table 4.3 for the sequential execution times).

Application	Blizzard/T	Blizzard/E	Blizzard/S	Blizzard/ES
appbt	6.3	2.7	4.2	4.8
barnes	6.9	4.4	4.4	5.3
em3d	3.8	2.2	3.6	2.7
lu	10.0	4.4	5.8	8.0
ocean	8.4	4.7	7.2	6.5
moldyn	5.5	2.6	4.5	4.0
tomcatv	10.6	8.7	6.8	9.0
waterns	11.5	10.5	8.1	9.9
watersp	12.1	9.9	8.2	10.9
appbt-cs	10.1	9.8	6.3	9.0
em3d-cs	17.8	16.5	12.6	17.1
moldyn-cs	7.9	6.3	5.9	6.7

col and the messaging subsystem that provide additional evidence for the interaction of the applications with the FGDSM systems.

4.3.1 Transparent Shared Memory Performance

Figure 4-1 presents the application execution times normalized to the execution time with Blizzard/T. Moreover, each execution time is broken down into three components:

- Computation time. It is the time spent in the computation. It includes the FGDSM overhead that directly affects the computation (polling code and access checks for software fine-grain tag implementations). Also, it includes the access fault overhead.
- Protocol time. It is the time spent running protocol (access fault and message) handlers. It also includes the time spent waiting a miss to be resolved.

- Synchronization time. It is the time spent waiting to acquire a lock or pass a barrier. However, it does not include informal busy-waiting synchronization (i.e., not through lock and barrier primitives in the user protocol libraries). Such synchronization appears in *appbt* and it is charged as computation time.

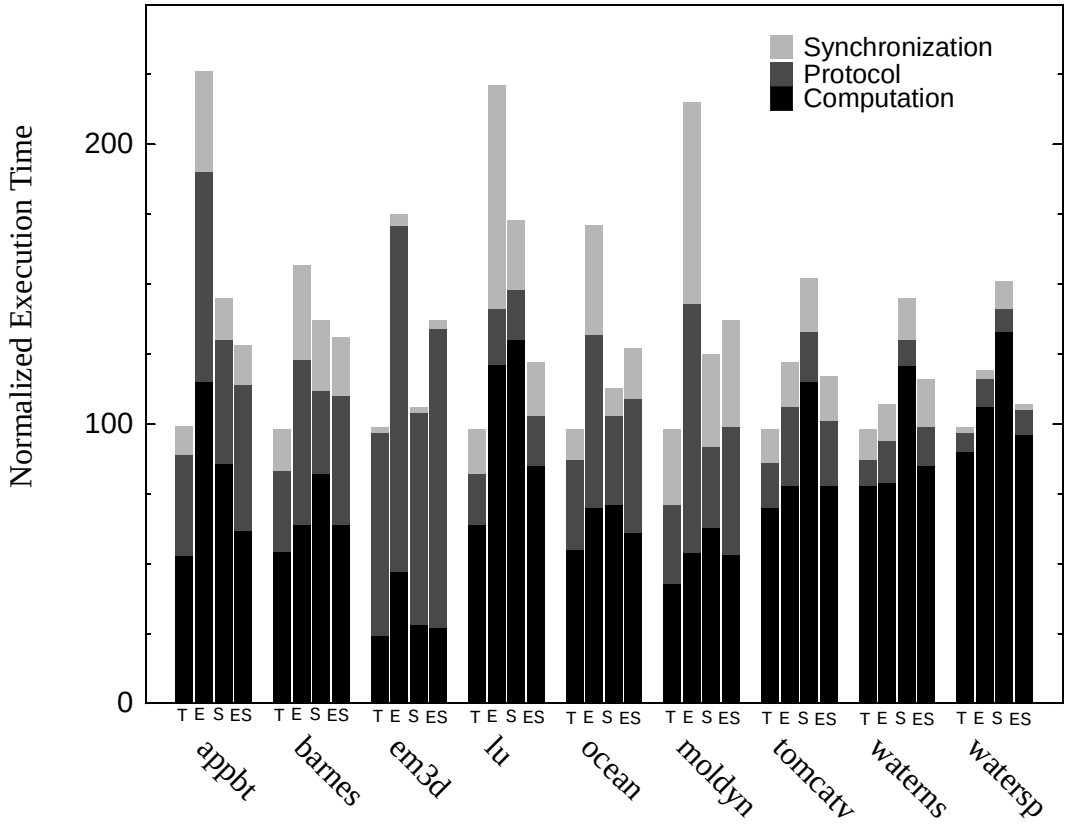

Figure 4-1. Transparent shared memory application performance (*stache*).

The figure presents the application execution times for the TSM applications normalized against the execution time with Blizzard/T. The execution times are broken down into three components: time spend in the computation (including polling instrumentation and access fault detection overhead), time spend in message handlers or access fault handlers or while waiting for a miss and synchronization time (barriers and locks).

Blizzard/T's results indicate that the application speedups are directly related to the protocol and synchronization time, which serves as a validation for the measurements. With Blizzard/E as expected, the protocol time increases relative to Blizzard/T due to the higher cost of the fine-grain access control mechanisms. It is not surprising that the increase in the protocol time exaggerates load imbalances in the applications and therefore, the synchronization time also increases. However, the dramatic increase in the computation time by more than 100% for some applications, shows that using page protection to distinguish read-only from read-write

blocks can have adverse performance effects. Performance suffers even in the absence of sharing, if there are a large number of writes to read-only pages (i.e., pages with at least one read-only block) in the application. As an example, *lu*, which achieves reasonable speedups with Blizzard/T, exhibits poor performance with Blizzard/E.

Blizzard/S's performance results do not show any surprises. The computation time increases due to the access checks in the application code, the synchronization time slightly increases as load imbalances exaggerated and the protocol time slightly decreases due to the lower overheads in manipulating the fine-grain tags.

Blizzard/ES strikes a balance between Blizzard/E and Blizzard/S offering the best overall performance. All three time components are consistently closer to Blizzard/T than with any other fine-grain tag implementation. Compared to Blizzard/E, Blizzard/ES addresses the page protection problem by performing tag lookups for stores in software and thereby obviating the need for read-only page protection. Compared to Blizzard/S, it addresses the access check problem by only inserting the check code for stores, which are less frequent than loads.

Overall, the performance results suggest than hardware acceleration for fine-grain access control offers superior performance. In its absence, a combination of software and commodity hardware methods performs better than either software or hardware alone. If combination methods are not feasible then software mechanisms are preferable to incomplete hardware ones.

4.3.2 Application-Specific Protocol Performance

The conclusions of the previous subsection change when we examine the CS applications. Figure 4-2 presents the application execution times for the three applications (*appbt*, *em3d*, *moldyn*) that there are both TSM and CS versions. All the execution times are normalized to the execution time of the TSM version with Blizzard/T. Furthermore, the execution times are broken down more finely than previously into five components:

- Computation time. Same as previously.
- Message handler time. It consists of the time spent running message handlers when message arrivals are detected through polling.
- Miss Time. It consists of the time spent executing access fault handlers and waiting for a miss to be resolved.
- Computation send time. It is the time spend sending messages directly from the computation, an ability used by custom protocols.
- Synchronization time. Same as previously.

For all fine-grain tag implementations, CS protocols eliminate the miss time and considerably reduce the message handler time. Indirectly, the synchronization time is also reduced. Since they send messages from the computation, the message send time is added to the total time but it is a fraction of the total savings. Moreover, the computation time is slightly reduced because no or few access faults exist and the access fault overhead is included in the computa-

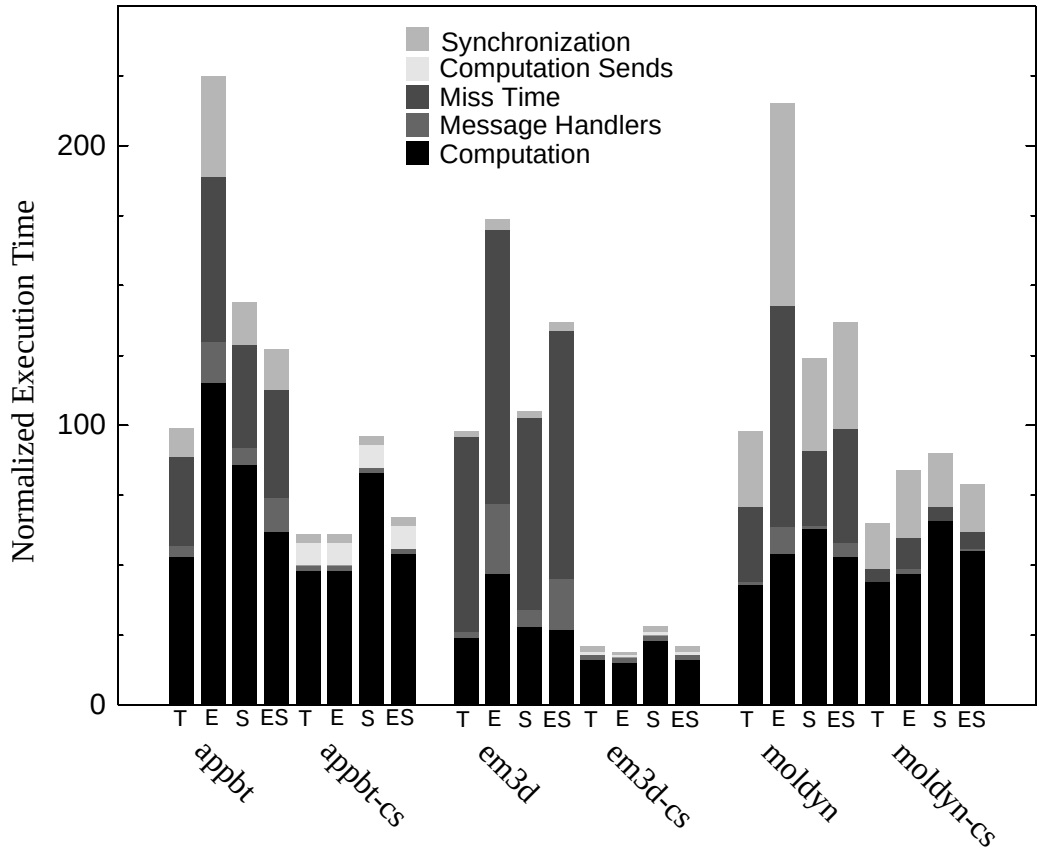

Figure 4-2. *Stache* vs. application-specific protocol performance.

The figure presents the application execution times for the TSM and CS versions normalized against the TSM execution time with Blizzard/T. The execution times are broken down into five components: time spend in the computation (including polling instrumentation and access fault detection overhead), time spend in message handlers invoked through polling, time spend while executing an access fault handler or waiting for a miss, time spend sending messages directly from the computation and synchronization time (barriers and locks).

tion time. The performance gains in the computation time are especially pronounced for Blizzard/E due to the cost of emulating the readonly state with the page protection.

These results shed new light to the performance comparison of the fine-grain tag implementations. If the overhead of the fine-grain access control is removed, Blizzard/E offers performance competitive to Blizzard/T's. Blizzard/S however, is remains affected by the access check overheads and fails to fully take advantage of the performance benefits of custom protocols. Blizzard/ES performs slightly worse than Blizzard/E. If this slight performance reduc-

tion is acceptable, in the absence of custom hardware acceleration for fine-grain tags, combination methods remain preferable. Otherwise, the difficulty to develop custom protocols must be judged against the reduced performance potential of Blizzard/S. If good performance for TSM programs is more important than good performance for CS programs software methods are preferable. If the opposite is true and the development time for custom protocols is not a concern, imperfect commodity hardware methods are preferable.

4.3.3 Protocol and Message Statistics

Table 4.5 presents low-level statistics gathered by the stache protocol during the execution of the applications on sixteen nodes with Blizzard/T. Stache keeps track of read faults and write faults and messages sent by the home nodes (invalidate requests) or the stache nodes (data and upgrade requests). The statistics do not differ significantly for other fine-grain tag implementations since these applications are static codes and their access patterns are not dependent on the timing of the protocol events.

The results reveal details on the protocol events that are caused by the application and therefore, the application sharing patterns. Moreover, they can be used to evaluate the data distribution in the application. Stache performs better when one of the sharing nodes for a particular block is also the home node for that block. For example, when two nodes share a block, stache is a two-hop protocol since every request generates only the reply. In contrast, when the home node is not one of the two sharing nodes, stache is a four-hop protocol since the requests and the replies are forwarded through the home node (see Chapter 1).

For example, it is immediately obvious that the sharing patterns in *lu* produce a massive transfer of readonly blocks from the home node to other nodes. Only read faults are generated and each one of them results in a data request. Similarly, *em3d*, *tomcatv* and partially *appbt* are dominated by a producer-consumer relationship where the home node writes new values that are consumed by another node. *Watersp* exhibits a similar pattern but this time more than one other nodes are involved in reading the data. The sharing patterns for the other TSM applications (*ocean, moldyn, waterns*) appear to be more complicated. *Moldyn-cs* shows widely-shared data (few write faults cause a lot of invalidations). Finally, the other CS applications do not use stache at all (*appbt-cs, em3d-cs*).

Table 4.6 presents low-level statistics gathered by the messaging subsystem during the execution of the parallel application on sixteen COW nodes with Blizzard/T. The messaging subsystem keeps track of small and large messages as well as the number of bytes transferred. In TSM applications, small messages carry one to three data words while large messages carry 36 words of data (a 128-byte stache block and three word arguments). In CS applications, large messages carry different sized data blocks. *Appbt-cs* sends messages that contain five doublewords. *Em3d-cs* and *moldyn-cs* packetize data values in 4-Kbyte messages. In addition, in *moldyn-cs*, the stache block size is increased to 1024 bytes.

Table 4.5: Stache statistics.

The table lists stache protocol events as observed during the execution of the parallel applications on sixteen COW nodes under Blizzard/T. All the statistics have been averaged by the number of nodes. The table includes the read and write faults and the request messages sent by the fault handlers. It also includes the invalidate messages sent by the home nodes. It does not include replies to the request messages from the home nodes since with very rare exceptions there is one-to-one correspondence with the request messages.

Application	Read Faults	Write Faults	Upgrade Msgs	Get RO Msgs	Get RW Msgs	Inval RO Msgs	Inval RW Msgs
appbt	31505	24439	4978	30093	559	24790	5140
barnes	6148	2332	1633	6093	7	4265	1624
em3d	202752	101376	0	202752	0	202752	0
lu	12639	0	0	12516	0	0	122
ocean	13831	10786	2179	10838	422	8673	2589
moldyn	37968	19528	16131	34247	180	18335	16233
tomcatv	12656	12072	185	12281	93	11973	281
waterns	11577	4608	4142	10851	0	4647	4146
watersp	2711	523	10	2719	0	1803	10
appbt-cs	2	0	0	0	5	0	1
em3d-cs	0	0	0	0	0	0	0
moldyn-cs	3869	856	824	3609	214	2816	1026

The table also includes density metrics, computed by simply dividing the message statistics with the total execution time. While the raw message statistics do not change with other fine-grain tag implementations, the density metrics depend on the execution time, which varies for every tag implementation. The byte density for TSM applications directly correlates with the application speedups. Data are transferred on shared memory misses and, therefore, more messages means more FGDSM overhead. Comparing the message statistics for those applications that both TSM and CS versions exist, we verify that the CS versions move more data with fewer messages both in total and per unit time.

Table 4.6: Message Statistics.

The message statistics have been computed by running the applications on sixteen COW nodes under Blizzard/T. All the statistics have been averaged by the number of nodes. The table lists the total number of small (control) and large (data) messages exchanged. It also lists the total number of message bytes exchanged. The last two columns contain the number of messages and bytes transferred during the execution of the parallel application divided by the total execution time.

Application	Control Messages	Data Messages	Message Bytes	Message Density (msgs/sec)	Byte Density (bytes/sec)
appbt	95924	35066	4628828	7457	285344
barnes	25727	7702	1016746	6982	235567
em3d	608634	202752	26763264	18727	673859
lu	12890	12639	1668381	3411	229684
ocean	36361	13851	1828390	6851	269022
moldyn	121077	50660	6687186	7413	309291
tomcatv	37736	12656	1670625	3620	130637
waterns	41318	14997	1979620	2365	90571
watersp	6428	2721	359197	1691	71111
em3d-cs	6690	4764	19204119	1233	2070409
appbt-cs	24	48243	2475675	4387	225004
moldyn-cs	14176	7186	12722073	1318	788112

4.3.4 Evaluating Fine-grain Access Control Alternatives

The results in this section allow us to evaluate the performance of the different fine-grain tag implementations against the design and implementation costs of the alternative techniques. Blizzard/T provides always the best performance. Blizzard/T's costs include the design cost of a custom board and the programming effort to properly support it inside the operating system. Blizzard/E displays inconsistent performance results ranging for very close to Blizzard/T to significantly slower than any other fine-grain tag implementation. Blizzard/E's main cost

component is the programming effort to implement the appropriate kernel support for the technique. The amount of effort required is only moderately larger than with Blizzard/T. Blizzard/S's performance results were always consistent, but also 30-70% slower than Blizzard/T. However, Blizzard/S design costs do not include hardware design or kernel-level programming. Instead, our efforts are focused in implementing a tool to insert the access checks before memory operations (e.g., binary rewriting tool or compiler). Blizzard/ES gives the best results without a custom hardware board (only 9-30% slower than Blizzard/T). However, its design costs includes both the kernel support for the ECC technique and the tool to insert the access checks.

This overview allows us to plan a path towards porting Blizzard on a new platform. Given the consistent performance and ease of implementation, Blizzard/S should be the first version. Then, if hardware fine-grain access control mechanisms such as the memory controller's ECC are available, we can consider using them in hybrid systems like Blizzard/ES to reduce the lookup overhead. In this way, we will also implement the appropriate kernel support for hardware fine-grain access control techniques. Eventually, if the extra performance gain of a custom fine-grain access control technique is justified by the extra design costs, we can consider custom-built boards for fine-grain access control acceleration.

4.4 Blizzard Evaluation

In this section, first I use a simple performance model to demonstrate the effect of the relative performance of local to remote memory references on the application performance. Subsequently, I contrast the performance characteristics of Blizzard to other shared memory platforms.

4.4.1 Remote Memory References and Application Performance

A simple performance model can be employed to demonstrate the importance of the performance ratio of remote to local memory references for parallel applications. The model does not claim to predict the performance of parallel applications on a specific platform. Rather, it is a first approximation to quantify the importance of remote memory references.

Suppose that we have a uniform memory architecture platform (UMA). In this machine, we cannot distinguish between local and remote references. Therefore, the performance ratio of remote to local references is exactly one. This machine corresponds to symmetric multiprocessor architectures without any data caches. We can break the total execution time T_P of parallel applications into three components:

- Computation time T_C that consists of the time spent in all the operations that do not access the memory.
- Local memory access time T_L that consists of the time spent in accesses to data private to each processor.
- Remote memory access time T_R that consists of the time spent in accesses to data shared between the processors.

Then the following formula obviously holds: $T_{UMA} = T_C + T_L + T_R$.

Suppose that due to technology or cost constraints, we cannot build a UMA machine. Instead, we are forced to make remote memory references *n* times slower compared to local ones. Effectively, we have built a machine with a non-uniform memory architecture (NUMA). Compared to the UMA machine, the new machine executes the same parallel programs in time: $T_{NUMA} = T_C + T_L + n * T_R = T_{UMA} + (n - 1) * T_R$. If the percent of the time that the parallel application spends on remote memory references on the UMA machine is *R*, then the expression can be written as $T_{NUMA} = T_{UMA} + (n - 1) * R * T_{UMA}$. To illustrate the cost of this design choice for parallel applications, we can compute the value of *R* for which the application performance is *k* times worse than in the UMA machine. Substituting *k* and *R* in the previous expression and solving for *R* yields that the application performs no worse than *k* times if $R < (k - 1) / (n - 1)$.

As an example, consider a parallel platform, similar to Blizzard/S, where remote memory references are two hundred times slower than local ones. Parallel applications on this machine will experience no more than 20% slowdown compared to the UMA machine, if they would have spend less than 0.1% of the execution time in remote references on the UMA machine. Therefore, if no technology constraints dictate otherwise, we would only prefer the NUMA machine, if the following statement is true: our applications spend no more than 0.1% of the execution in remote references on the UMA machine and the UMA machine is at least 20% more expensive than the NUMA machine.

As the model has been presented, it does not take into account processor or node caches. Symmetric multiprocessing (SMP) and cache-coherent NUMA (CC-NUMA) architectures use the processor caches to cache local and remote data. Processor caches make local or remote references as cheap as computation. SMP and NUMA systems can have additional caches shared by all the node processors. Cache-only memory architectures (COMA) use a portion of the local memory as node caches for remote data. In systems with node caches when accesses to remote data miss in the processor cache, the node cache (or local memory for COMA architectures) is accessed. If the data are not locally available however, then they must be fetched from a remote node. Node caches make remote references as cheap as local references.

The cache characteristics affect the importance of remote memory performance. For example, it may be worthwhile to make remote references twice as slow if the cache size can be increased so that the number of remote misses is reduced by 50%. In general, large caches do not always translate to reduced miss rates for remote data. Caches exhibit compulsory, capacity, conflict [Hil87] and coherence misses [Jou90]. Capacity misses occur when the cache cannot contain all the blocks and conflict misses occur when two blocks map to the same cache location. Changing the cache characteristics reduces (or even eliminates for some applications) capacity and conflict misses. Blizzard and other FGDSM systems [RLW94,SGT96] typically implement simple COMA-like [HSL94] software coherence protocols to maintain coherence across nodes. Since the local memory is used as a node cache, in practice, capacity

and conflict misses are eliminated or transformed to local misses. In contrast, compulsory and conflict misses directly correspond to the characteristics of the parallel applications. Remote compulsory misses occur the first time remote memory location is accessed. In DSM systems, the number of capacity misses can be reduced by careful initial placement of the data. Coherence misses depend mainly on the application sharing patterns and indicate true communication requirements in the underlying algorithms. Therefore, they cannot be avoided and the application performance directly relates to the applications characteristics and how fast the remote misses can be serviced on a specific platform.

Conceptually, caches can be included in the model, if we define local and remote memory references as follows:

- Remote memory references are the references that access remote data, miss in the processor or node caches and data are brought from remote nodes.
- Local memory references are the references that access local or copies of remote data and miss in the processor cache.

While this change incorporates caches in the model, the lack of realistic models for the cache behavior of parallel programs does not usually allow to analytically predict the performance of parallel applications.

Moreover, an important shortcoming of the model is that it ignores the effects of synchronization. Slowing down remote memory references can increase the time that processors spend synchronizing because it exaggerates load imbalances in the application. For example, suppose one processor performs ten remote memory references and another one performs only five before a synchronization point. Then, the time that the second one wastes waiting for the first one to reach the synchronization point is proportional to the cost of the remote memory references. The relationship between the overhead of remote references and synchronization idle time is non-linear and it depends on the application sharing and synchronization patterns. For this reason, the evaluation of parallel platforms has been traditionally done with benchmark suites that correspond to real-world workloads. However, the model is still useful for its intended purpose to demonstrate the importance of remote memory performance and it can be used to provide a lower limit on the application slowdown with slower remote memory references.

4.4.2 Blizzard vs. Other Approaches

Blizzard's performance cannot be evaluated in isolation. Instead, it should compared to other shared memory approaches. Hardware shared memory systems offer excellent performance but require additional hardware. In loosely-coupled hardware distributed shared memory (DSM) systems such as Sequent's STiNG or Convex Exemplar X, remote memory references are five to ten times slower than local ones [Cor94,LC96, LL97]. More tightly-coupled DSM systems such as SGI's Origin [LL97], exhibit even better performance characteristics. Remote memory references are only two times slower than local ones. Symmetric multiprocessors with aggressive interconnection networks such as Sun's ES 6000, support a UMA model where no distinction exists between local and remote memory references. Clearly, Blizzard does not belong in this category. However, these examples are high-end

servers with significantly higher hardware costs compared to the COW platform. Latency-limited applications benefit from the hardware support and justify the premiums charged for such platforms.

Previous software distributed shared memory (DSM) systems relied on virtual memory page protection hardware to detect and control shared references. Sequentially consistent page-based software DSM systems such as Ivy [LH89] exhibit latencies as much as an order of magnitude higher than Blizzard. However, they typically achieve higher remote memory bandwidth since messaging hardware is optimized for large messages. Unfortunately, the large size of memory pages typically causes excessive false sharing and fragmentation, which limits their application domain to coarse grain applications [CDK+94, SFL+94].

Later software DSM systems such as TreadMarks [KDCZ93], mitigate the effects of false sharing and fragmentation with relaxed memory consistency models. Studies have shown that they can offer competitive performance to sequentially consistent software FGDSM systems [ZIS+97]. However, weaker consistency models complicate introduce complexity in the software design. With the exception of the Alpha and PowerPC architectures, most hardware implementations of shared support simple consistency models. Moreover, Hill argues that future hardware implementations of shared memory should support simple consistency models [Hil97] because relaxed consistency models do not offer a significant performance advantage to justify the extra software complexity. Introducing relaxed consistency models specifically for software DSM systems, complicates the porting of shared memory applications to these systems.

FGDSM systems like Blizzard and Digital's Shasta can also support relaxed consistency models but the performance benefits are not significant [SGT96, ZIS+97]. Reported application performance results for Blizzard and Digital's Shasta demonstrate that FGDSM systems can achieve significant speedups for many scientific applications. Both these systems achieve comparable speedups for shared memory applications despite hardware and software differences between the two systems [SGT96,SFH+97].

Unique among FGDSM systems is Blizzard's ability to harness the raw messaging performance and offer it to the application. The Tempest interface [Rei94] exposes the access control and messaging mechanisms to user level and therefore, allows the development of application-specific coherence protocol [FLR+94]. Such protocols eliminate the access control overhead resulting in lower latencies for small data transfers. Moreover, if the applications are bandwidth and not latency limited, they can batch data transfers to large messages that offer remote bandwidth competitive with hardware shared memory systems.

4.5 Conclusions

In this chapter, I evaluated Blizzard as a platform for shared memory parallel applications. FGDSM systems like Blizzard follow a "middle of the road" approach to shared memory compared to hardware and sequentially consistent page-based software shared memory in

terms of performance. Software page-based shared memory with relaxed consistency models can offer performance similar to FGDSM shared memory but it complicates the programming model. Moreover, unlike other FGDSM systems Blizzard allows the development of application-specific coherence protocols which optimize data transfers to match the applications sharing patterns eliminating fine-grain access control overheads.

I presented low-level measurements of the remote and local memory performance. Subsequently, I focused on application performance. I used a benchmark suite that consisted of nine scientific shared memory applications. For three of the applications, the suite included alternative versions for which application-specific protocols have been developed. I presented application performance results, first for the transparent shared memory versions and then for the custom protocol versions under different fine-grain tag implementations.

The results demonstrated that Blizzard can achieve reasonable speedups but for applications with high communication requirements custom protocols are required. Moreover, custom hardware acceleration for access control offers superior performance in either case. In its absence, imperfect commodity hardware tag implementations are preferable if custom protocols are targeted while software tag implementations are preferable for transparent shared memory programs. Combination techniques offer the best tradeoff and should be pursued if possible.

Finally, I presented a simple model that was used to demonstrate the importance of remote memory performance. Subsequently, Blizzard’s performance was contrasted to other approaches to shared memory.

Chapter 5

Blizzard On Multiprocessor Nodes

A distinguishing characteristic of the Blizzard implementation on the Wisconsin Cluster of Workstations (COW) is its support for symmetric multiprocessor (SMP) nodes. FGDSM systems like Blizzard are particularly attractive for implementing DSM on SMP clusters because they transparently—i.e., without the involvement of the application programmer—extend the SMP's fine-grain shared-memory abstraction across a cluster.

Grouping processors into SMP nodes allows them to communicate within an SMP node using fast hardware shared-memory mechanisms. Multiple SMP processors can also improve performance by overlapping application's execution with protocol processing—i.e., handling FGDSM messages. Simultaneous sharing of node's resources (e.g., memory) between the application and protocol, however, requires mechanisms for guaranteeing atomic accesses [SGA97]. Without efficient support for atomicity, accesses to frequently shared resources may incur high overheads and result in lower SMP-node performance. FGDSM designs vary in the degree of support for atomic accesses, from zero-overhead custom hardware implementations providing atomic accesses directly in hardware [RPW96] to low-overhead synchronization operations in software [SFH+96,SGA97].

Performance results presented in this chapter, indicate that grouping processors in SMP nodes benefits from custom hardware support. Custom hardware support for fine-grain sharing significantly reduces synchronization overhead in SMP nodes. Furthermore, grouping processors in SMP nodes is beneficial for FGDSMs with high-overhead protocol operations. By converting FGDSM protocol operations (across nodes) to SMP hardware coherence operations (within node), grouping processors in SMP nodes boosts performance in FGDSM systems with high-overhead protocol operations. In pathological cases, ad hoc scheduling of

protocol operations on an SMP node can hurt performance due to adverse cache effects. In Blizzard, an idle processor on a node can handle protocol operations on behalf of another. As such, data that belongs to one processor may be received by others. Protocols which synchronously communicate large amounts of data may result in poor cache performance.

Overall, using SMP nodes results in competitive performance compared to the uniprocessor implementation while substantially reducing hardware requirements (nodes, network interface cards and switches/routers) in the system. Although, the manufacturing cost of computer products is typically related to cost of components, cost from the perspective of a customer is related to price which is also dictated by market forces [HP90]. In a cost-performance study related to the work presented in this chapter, we found that dual-processor nodes were more cost-effective in that specific timeframe than either uniprocessor or quad-processor nodes [SFH+97].

The rest of this chapter is organized as follows. Section 5.1 describes the mechanisms and system resources required to implement FGDSM on a node of a parallel machine and discusses the alternative policies for each resource. Section 5.2 summarizes Blizzard's choices for each resource and their rationale. Section 5.3 presents how Blizzard implements fine-grain tags on an SMP node. Section 5.4 discusses the factors affecting performance of SMP-node implementations of FGDSM. Section 5.5 analyzes the interaction between grouping processors in SMP nodes and protocol traffic. Section 5.6 evaluates the performance of SMP-node implementations. Section 5.7 discusses related work in software DSM systems for SMP clusters. Finally, Section 5.8 concludes with the summary of this chapter.

5.1 FGDSM Resources & SMP Nodes

In Chapter 1, we discussed the organization of an FGDSM system and identified the FGDSM system resources. The same resources also exist in SMP-node implementations. Shared data pages are distributed among the nodes, with every node serving as a designated *home* for a group of pages. A *directory* maintains sharing status for all the memory blocks on the home nodes. A *remote cache* serves as a temporary repository for data fetched from remote nodes. A set of *fine-grain tags* enforce access semantics for shared remote memory blocks. Upon access violation, the address of the faulting memory block and the type of access are inserted in the *fault queue*. Processor(s) are responsible for running the application and the software coherence protocol that implements shared memory. The protocol moves memory blocks among the nodes of the machine using a pair of send and receive *message queues*. The application can directly send messages and therefore, accesses the send message queue. This is typically used in application-specific protocols.

In an SMP environment, a policy is required to coordinate accesses to these resources by the application and the protocols handlers. There are two general policies that we can follow. The first one is to share the resource among the processors. However, this implies that we must pay the synchronization cost when accessing that resource. The second one is to replicate/distribute the resource for every processor. In this way we can avoid the synchronization cost.

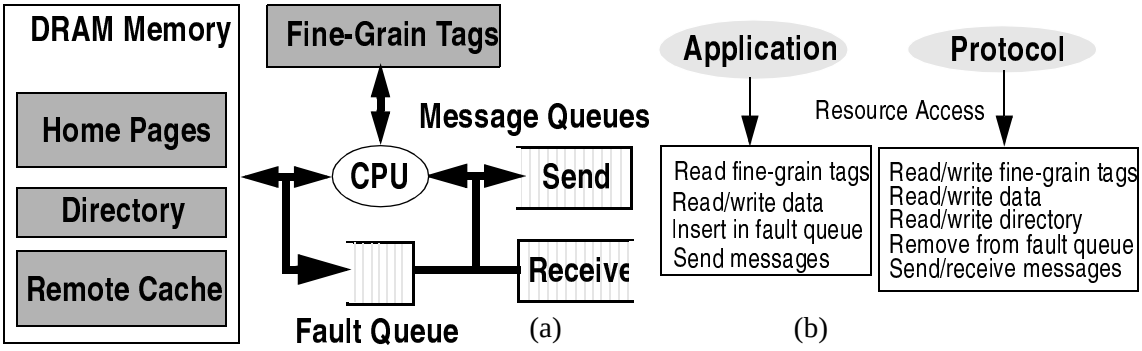

Figure 5-1. Anatomy of a software FGDSM node:
(a) resources and mechanisms in software FGDSM,
(b) breakdown of accessing resources between application and protocol.

However, we introduce extra memory demand and/or extra processing cost to maintain the copies consistent. In the rest of this section, I will focus on each FGDSM resource and discuss how these general policies are specialized in each case.

Fine Grain Tags & Remote Cache

The easiest way to port an FGDSM system on SMP nodes is to replicate all the resources for every processor using the hardware shared memory only to speedup intranode messaging. The advantage of this design is that all the changes required to support SMP nodes are localized in the network subsystem. However, it is a bad design because the remote cache, the first resource we should consider sharing, is also replicated. Sharing the remote cache has two important advantages that we should not overlook. First, when all the node processors access the same remote data, we reduce the memory demands of the program. Second, we can directly take advantage of the hardware shared memory for sharing among the node's processors without involving the FGDSM system.

Even if the remote cache is shared, this does not necessarily imply that the fine-grain tags must be shared. Depending on the tag implementation, it may possible to replicate them. In Chapter 2, we saw that the fine-grain tags can be associated either with the physical or the virtual address space. If the fine-grain tags are associated with the physical address space, there is no alternative to sharing. If the fine-grain tags are associated with the virtual address space, we can consider replicating them.

An application's shared-memory references require a tag lookup to enforce shared-memory access semantics—i.e., to determine whether the reference results in an access violation. The tag lookup and memory access must appear atomic to prevent a simultaneously running protocol handler from stealing the data. Therefore, a synchronization mechanism between the protocol handler and the application is required. Replication allows selective synchronization with only those processors that actually have accessed the remote data. The first time a processor accesses a block a trap is always delivered. If the block is local because some other processor has accessed in the past, then we need to simply update the fine-grain tags and continue. When a protocol handler wishes to downgrade the fine-grain protection, it will only need to synchronize with those processors that they have accessed the block in the past. Similar techniques have been used to reduce the cost of TLB shootdowns in SMP machines [ENCH96]. An FGDSM system that implements replicated fine-grain tags is Shasta/SMP [SGA97]. In Section 5.3, we will examine in detail how the different Blizzard tag implementations address the atomicity problem.

Message & Access Fault Queues

Running on an SMP node offers the opportunity to support separate message queues per processor. Such a policy allows the processors to access the message queues without any synchronization overhead. Moreover, replication allows us to direct messages to a particular processor on the node. On the other hand, replication does not come free. When message queues are replicated, there is an opportunity cost because messages cannot be serviced even if another processor is idle as well as a processing cost to figure to which particular processor a message should go.

In order to support replication efficiently, we need network interfaces that can efficiently service separate output queues and demultiplex incoming messages to separate input queues. For example, network interfaces that support a remote memory access model can trivially support separate message queues [BLA+94, Sco96] by building separate circular message queues in main memory. Incoming messages are directed to the appropriate memory location and therefore, the appropriate message queue on the remote node.

Sharing introduces extra overheads to synchronize competing accesses to the message queues. It is the only alternative with restricted interfaces that support strict queue semantics such as the CM-5 network interface [Thi91]. Since only one processor can access the message in the top of the queue, message operations need to be serialized and replication is not an option.

However, when the hardware allows random queue entry addressing, the extra sharing overhead is relatively small. Random queue entry addressing allows processors to synchronize only around the manipulation of queue pointers to enqueue or dequeue messages. Moving the actual message data, which consists the bulk of the time in sending or receiving messages, can occur concurrently. Such is the case with network interfaces that support remote memory accesses or even simply implement software message queues in onboard user-accessible

memory [BCF+95]. Message data can be accessed through memory pointers with normal memory accesses even after the queue head pointer does not point to the message.

Executing handlers requires a synchronization model for accesses by concurrent handlers to FGDSM resources. Two alternatives are atomic handlers or protocol locks. Atomic handlers effectively bypass the synchronization problem by allowing only a single handler in any particular moment to be active throughout the node. This is a very simple model with minimum overheads and complexity for the protocol code. However, it does not scale to a large number of processors per node. As the number of processors per node increases, the probability that more than one handler will be available for execution at any particular instance also increases.

Protocol locks are simply locks in local memory that are used by the handlers to synchronize accesses to shared resources. Supporting protocols locks, however, is more complicated that one would expect. The complication is introduced because the execution model tries to map an infinite number of handlers to a finite number of processors. This introduces a deadlock condition as follows. Suppose all the processors start executing protocols handlers and all the protocol handlers block waiting on some lock. A message arrives that will should the dispatch of a handler to release the lock. However, since all the processors are busy with the handler, the new protocol handler can not be dispatched. There are three alternative ways to break the deadlock. The first one is to include the degree of concurrency (number of processors per node) in the model, which limits portability across SMP nodes with different number of processors per node. The second one is to provide restartable handlers. This is very difficult to do transparently to the protocol code while requiring the handlers to rollback if an attempt to lock fails introduces complexity. Finally, the third one is to provide a restricted synchronization model that allows only mutual exclusion and not conditional synchronization [AS83]. This can be achieved by constraining the protocol handlers to release all the locks they acquired before they are allowed to finish. This last scheme looks quite promising for shared memory protocols since they have modest synchronization requirements.

The fault queues are conceptually similar to the input message queue. However, they are not tied with the network architecture. On a uniprocessor node, the fault queue may simply be procedure parameters passed in processor registers in a protocol call. On an SMP node, the fault queue may be a per-processor array of fault records or an actual queue in hardware that a protocol thread polls on. The location of the fault queue can be in registers, memory or even device registers. Fault handlers need to be synchronized with respect to message handlers. Once the synchronization model for message handlers is decided, the same one should be used for miss handlers too.

Directory & Home Pages

In typical shared memory protocols, the directory represents a single point of synchronization that is responsible for enforcing the coherence protocol among the nodes. Typically, the responsibility of managing the directory is distributed among the nodes. FGDSM systems tend to have simpler directory schemes than their page level DSM cousins because they do not

have to rely on sophisticated consistency models for acceptable performance [LH89,ZIS+**97**]. In this aspect, they resemble hardware directory schemes. In fact, the protocols usually running on such systems have been modeled out of hardware shared memory protocols. Typically, each node is assigned the management of a particular address range in a simple way (e.g., round robin).

With SMP nodes, the question is whether the responsibility for the portion of the shared address space managed by the node should be shared or distributed among the node's processors. In general, sharing offers an opportunity for better load balancing because an idle processor can service requests for any part of the address space with the potential danger of more expensive synchronization.

The directory management policy is tied with the policy for incoming message queues. With replicated message queues, the ability to address each individual processor, allows us to directly extend the directory scheme to SMP processors. The downside is that we will not be able to take advantage of idle processors to service remote requests. Similarly, it does not make sense to maintain separate directories with a single shared message queue because there is no easy way to direct messages to processors. For example, with a shared queue, you cannot guarantee that an incoming directory request message will be executed on the particular processor that manages the appropriate directory.

5.2 Blizzard Implementation Overview

In Blizzard, processors share all of the FGDSM resources, such as the fine-grain tags, remote cache, home pages, directory, and fault and message queues. Sharing the remote cache allows processors to eliminate replication by maintaining a single copy of a remote data on a node. Sharing the fine-grain tags is used for two reasons. First, Blizzard fine-grain implementations include hardware methods in which fine-grain access control is tied to the physical address space and therefore, it cannot be replicated. Second, pure software fine-grain implementations, which can choose to replicate the fine-grain tags, do not require because COW nodes support up to four processors only obviating the need for replicated tags.

Sharing the message queues allows for a better utilization of processors by overlapping computation on some processors with protocol processing on others. Furthermore, we avoid extra overhead on the network interface to service multiple message queues. Sharing resources, however, often requires atomicity of accesses. Without hardware support for atomic operations, accesses to shared resources may incur high synchronization overhead and result in lower performance. Synchronization overhead is further pronounced if the shared resource is accessed frequently. As such, efficient mechanisms for guaranteeing atomicity of accesses to frequently shared resources are of critical importance.

Blizzard's synchronization model for protocol handlers is based on atomic handlers. Protocol handlers typically consist of code to move a fine-grain (32-128 byte) data block between memory and the message queues, update the corresponding directory state and tag, and

remove an entry from the fault array. Moving messages between memory and message queues often dominates handler running time on commodity networking hardware. Commodity network interface cards are typically located on slow peripheral buses with limited bandwidth of accesses from processors and/or memory. Furthermore, these cards typically do not provide support for simultaneous dispatch of multiple messages. Handler execution, therefore, is inherently a serial operation and parallelizing it would incur high software synchronization overhead, result in high protocol occupancies [HHS+95] and lower overall performance.

Blizzard serializes protocol handler execution on SMP nodes. Serializing handler execution obviates the need for synchronization of accesses for resources used only by the protocol, i.e., the message queues and the directory state. Accesses to fine-grain tags, remote cache and home pages, however, rely on mechanisms described in Section 5.3 for atomicity guarantees. The fault array also requires atomic accesses, as it may be simultaneously referenced by the application. Fault array accesses are atomic since fault information consists of an address and access type pair which is simply stored as a doubleword into per-processor designated storage.

Blizzard multiplexes computation with running protocol handlers on all the processors. Alternatively, the system can dedicate one processor on every node to execute protocol handlers. Various researchers have explored scheduling policies for commodity DSM clusters [KS96,ENCH96,FLR+94]. One such study [FW97,Fal97] examines scheduling policies for FGDSMs on SMP clusters and concludes that a dedicated processor is not advantageous for slow commodity networking hardware and small-scale SMP nodes.

In Blizzard, processors contend for a software lock to assume the role of the *protocol processor*, (a) upon an access violation, (b) while waiting at a barrier synchronization, or (c) upon arrival of a message. As we discussed in Chapter 3, the network interface card signals the arrival of a message by setting a flag in a user-accessible memory location. In the absence of a dedicated protocol processor, the application code is instrumented with executable editing, to poll on the flag on every loop-backedge. A protocol processor always goes back to computation by releasing the lock when it no longer has to wait—e.g., a remote block arrives or all the pending messages are received.

5.3 Fine-Grain Tags

FGDSM's most frequently accessed resources are the memory and the fine-grain tags. Fine-grain tags maintain shared-memory access semantics to the remote cache and home pages. Shared-memory references to the remote cache or home pages in the application (Figure 5-1 (b)), therefore, require a fine-grain tag lookup. On an SMP node, a protocol handler executing on one processor and the application running on another processor may be simultaneously accessing the fine-grain tags and memory. As such, shared-memory semantics dictates that accesses to the fine-grain tags and the corresponding data in memory execute atomically [SGA97,SGT96,SFL+94]. Mechanisms for atomic tag and memory accesses can vary depending on the implementation of the fine-grain tags.

Fine-grain tags and memory may be simultaneously accessed by the application and protocol handlers running on multiple SMP-node processors. An application's shared-memory references each require a tag lookup to enforce shared-memory access semantics—i.e., to determine whether the reference results in an access violation. The tag lookup and memory access must appear atomic to prevent a simultaneously running protocol handler from stealing the data (Figure 5-2). Because of their high frequency, tag lookups require efficient mechanisms to guarantee atomic accesses.

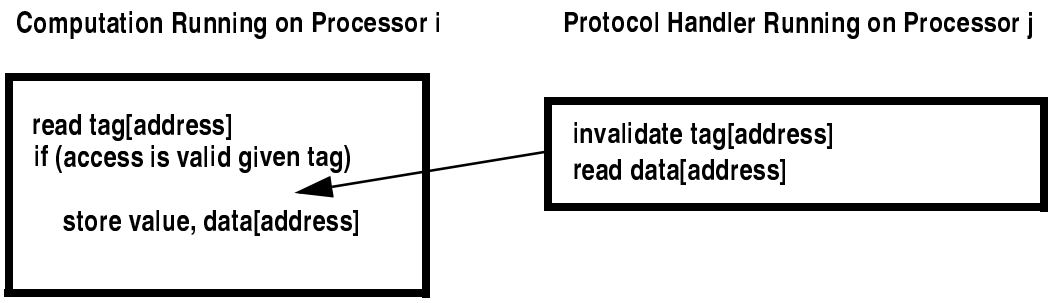

Figure 5-2. Example of race conditions in tag accesses.
FGDSM systems must avoid race conditions such as the one depicted in the figure. The protocol handler changes the tags and reads the data between the tag access and the store operation by the computation. As a result, the store value is lost, which violates shared memory semantics.

Protocol handlers also require atomic accesses to fine-grain tags and data in remote cache or home pages. An FGDSM protocol handler uses three types of operations to implement coherence. First, a protocol handler may place fetched remote data into the remote cache or home pages in memory. Such an operation requires writing data to memory while modifying the tag atomically. Second, a protocol handler may invalidate or downgrade (i.e., change to read-only) a read-write block in the remote cache or home pages. Such an operation requires reading data from memory while modifying the tag atomically. Third, a protocol handler may invalidate or upgrade (i.e., change to read-write) a read-only block in the remote cache and home pages. Such an access is simply atomic by nature and does not require special support.

Fine-grain tags may support atomic accesses from the application and protocol handlers either in hardware [RPW96, SFL+94] or software [SGT96, SFL+94]. Hardware implementations may store tag values in SRAM on a custom bus-based snooping board, or in error-correcting code (ECC) in the memory system. In the frequent case of tag lookup and data reference from the application, hardware implementations guarantee atomicity efficiently by performing the tag lookup in hardware at the time of the data reference. Software implementations store the tag values in memory and use executable editing to instrument shared-memory references to perform a tag lookup before the memory reference in software. Implementations

vary as to how atomicity is guaranteed for the less frequent accesses from protocol handlers. In this section, we will examine the various implementations of fine-grain tags in Blizzard and their support for SMP nodes.

Blizzard/T: Custom Board SRAM Tags

Blizzard/T uses Vortex [Pfi95], a custom board that snoops memory bus transactions, performs fine-grain access control tests through a SRAM lookup, and coordinates intra-node communication. The board contains two FPGAs, two SRAM, and some minor logic [RPW96]. Vortex enforces fine-grain access semantics by asserting a bus error in response to memory transactions that incur access violations—e.g., read/write to an invalid memory block or write to a read-only block. As we saw in Chapter 2, an optimized kernel vector trap table delivers the bus error to the user level [TL94a,RFW93,SFH+96]. Just before control is passed to the user-level handler, the kernel writes the fault information in the thread's private area. In a uniprocessor-node implementation, the user-level code retrieves the fault information and directly calls the appropriate protocol handler. In an SMP-node implementation, the protocol code polls in the appropriate location of the private thread areas, which effectively act as the fault queue.

Blizzard/T provides hardware mechanisms for protocol accesses to the fine-grain tags. By writing to a Vortex control register, handlers can atomically read a memory block while invalidating/downgrading the corresponding tag. Upon a write to the control register, Blizzard/T updates the tag and reads the data into a handler-accessible block buffer on the board. When placing fetched remote data into memory, handlers require an atomic write to memory and an update of an (invalid) tag. Instead of supporting atomic writes directly in hardware, Blizzard/T uses a mechanism to bypass hardware tag lookup while writing the data before updating the tag. Because protocol handlers in Blizzard always execute to completion without interruptions, the two non-atomic operations (i.e., the write to data and the tag update) appear to execute atomically with respect to the application. Access violations from the application during handler execution will be caught by the system and the application will be immediately resumed after the tag update. Blizzard/T bypasses tag lookups by writing to uncached aliased page mappings to memory.

Blizzard/S: Software Tags

Blizzard/S implements the tags in software by storing them in memory. Using executable editing [LS95], Blizzard/S inserts access control tests around shared-memory loads and stores in a fully compiled and linked program. Blizzard/S uses two forms of tests to detect access violations. Invalid memory is marked with a sentinel value that has a low probability of occurring in the program [SFH+96,SGT96]. The most common test case uses a sequence of 3 instructions (3 cycles) to detect word and doubleword load operations to invalid memory blocks. When the test detects a sentinel, it performs a complete table lookup in order to distinguish access violations from innocent uses of the sentinel value. The majority of the other

memory operations (i.e., all stores and some loads) use a sequence of 5 test instructions (6 cycles) to index a tag table prior to the memory reference to detect access violations.

Fortunately, similar to hardware implementations, sentinels do not require additional synchronization since the lookup is implicitely tied to the datra reference. Software tag table lookups however, use memory instructions and are not atomic with respect to data references. Uniprocessor-node implementations such as Shasta [SGT96] and Blizzard/S [SFL+94] eliminate the race between protocol handlers and the application by suppressing all protocol activity during the test; both systems only use instrumented polling rather than interrupts to invoke protocol code upon message arrivals. Because message polling code is only inserted at loop-backedges, protocol activity does not interfere with tag lookups. Therefore, the tag lookup and the data reference in the application appear to execute atomically.

Blizzard on an SMPs, however, overlaps execution of protocol handlers and applications among SMP-node processors. Blizzard/S guarantees atomicity of accesses to tags and memory by implementing a handshake between the application and the protocol handlers. The handshake is in the form of synchronization instructions in the instrumentation to a per-processor memory location that protocol handlers poll on (Figure 5-3). Synchronization in the application consists of a pair of store and clear instructions to a per-processor memory location encapsulating an instrumented memory operation. Upon invalidating or downgrading a block from read-write to read-only, the protocol handler can safely modify the tag in advance, but has to guarantee that all writes to the data from the application have completed. For each handshake, this scheme causes one miss on each processor other the one where the protocol handler is executed as well as one miss for each processor on the protocol handler.

Unfortunately, the handshake instrumentation overhead may be high (i.e., two stores per instrumentation) for applications with a large number of non-atomic instrumentations. Moreover, in applications with a small amount of protocol activity, synchronization with protocol handlers results in unnecessary overhead in the computation. In addition, forward progress may be an issue if the SMP coherence protocol implementation inhibits the protocol handler loop from observing rapidly alternating changes in the poll flag.

As a result of these observations, a second version of Blizzard/S, Blizzard/SB (B stands for backedge) implements a second scheme, which reduces the frequency of store and clear instructions and it does not depend on the coherence protocol for forward progress. Blizzard/SB inserts the store instruction inside the protocol handler after updating the tag, and inserts the clear instruction only in loop-backedges in the application (Figure 5-4). Blizzard/SB optimizes synchronization overhead in applications with low frequency of protocol activity while increasing the protocol waiting time in protocol-intensive applications. For each handshake, this scheme causes one miss on each processor other the one where the protocol handler is executed as well as two misses for each processor on the protocol handler. Furthermore, it delays the protocol handler by the average time it takes for the application code to encounter a backedge branch.

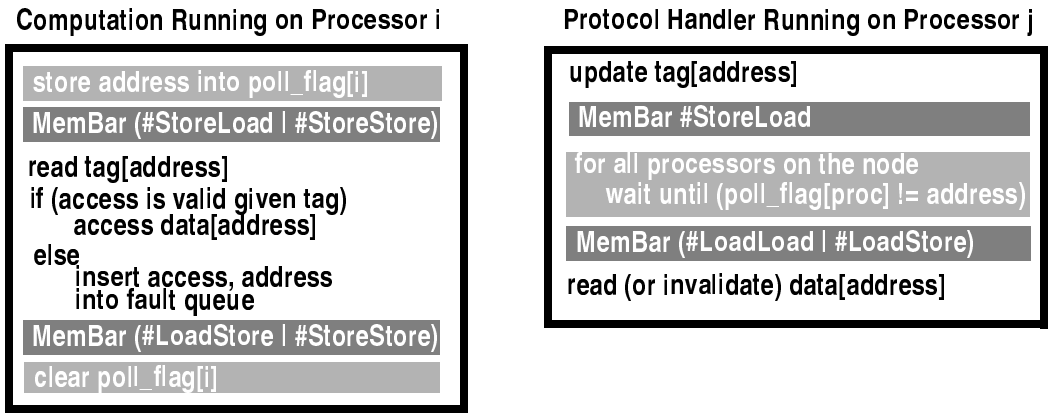

Figure 5-3. Software handshake in Blizzard/S.
The figure depicts the software handshake between the application (left) and protocol (right). Synchronization statements (shaded light gray) in the application consist of a store instruction indicating the block which the application is accessing and a clear instruction indicating there are no accesses in progress. A protocol handler can safely invalidate/downgrade the tag in advance, but has to guarantee that the application is not in the middle of accessing the data before the handler can read or invalidate it. Memory barriers instructions (shaded dark gray) are required for correctness with weaker SMP consistency models. I use Sparc V9's [WG94] memory barrier instructions to illustrate the ordering requirements with respect to weaker consistency models. These instructions bit-encode the ordering relationships. For example, the mask (#LoadLoad | #LoadStore) indicates that all loads before the memory barrier should complete before any other loads or stores issued by the same processor).

Blizzard supports a third handshake method based on increasing counters called epochs. In this scheme, the application increments the value stored in the per processor location in every backedge branch (Figure 5-5). The protocol handler waits until the value changes, which signals that the application has reached a backedge branch. For each handshake, the behavior of this scheme is similar with the previous one. Its advantage is that it substitutes a miss on an invalid block with a miss on an shared block. Therefore, the cache block containing the flag will not bounce between the application and the protocol handler. However, it requires one extra global register for the epoch counter or else the counter must be stored in memory and loaded on every backedge branch. Since an extra free register was not available on COW, the scheme failed to show any advantages over Blizzard/SB and I will not discuss it further.

All the synchronization techniques implemented in Blizzard assume a sequentially consistent system. Weaker memory models require additional memory barrier instructions to make

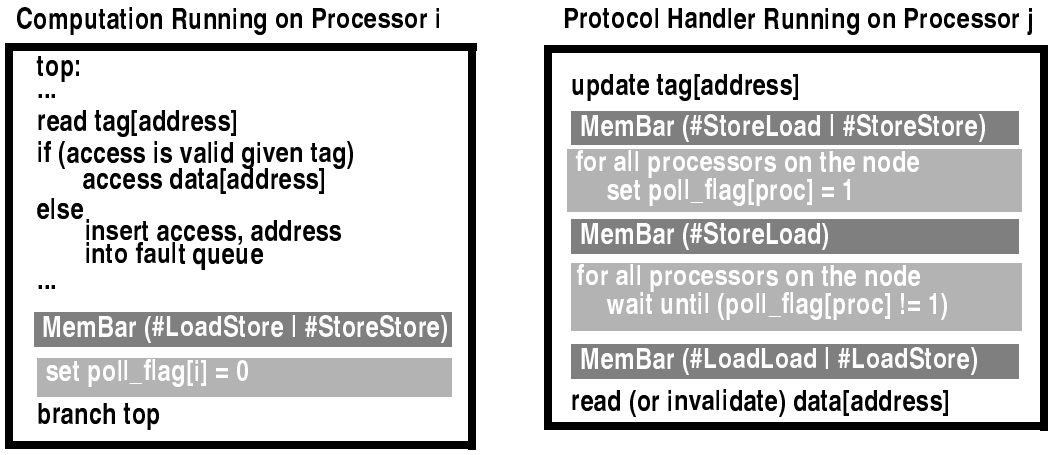

Figure 5-4. Software handshake in Blizzard/SB.
The figure depicts the software handshake between the application (left) and protocol (right). The sole synchronization statement (shaded gray) in the application is a clear instruction on loop backedges. The protocol handler sets the flags for all processors (except itself) and then waits until they are cleared. The SMP-node memory consistency model guarantees that the tag changes have propagated to all the processors. As in Figure 5-3 , memory barriers instructions (shaded dark gray) are required for correctness with weaker SMP consistency models.

the store and clear instructions visible to other processors. In the long run, this may be a relatively insignificant issue since Hill [Hil97] presents a convincing case that future multiprocessors should support simple memory consistency models. If however, we are targetting an SMP that only supports a weaker consistency model and the the cost of such instructions is prohibitively expensive, we can consider a fourth handshake method that has been implemented in Shasta/SMP [SGA97]. In the Shasta scheme, the protocol handler explicitly initiates a handshake event with a message to the other processors. Upon receipt of the handshake request, the processors have to reply with explicit response messages. In this way, no extra overheads other than the code already in place to poll for messages, are introduced in the application side. However, these local messages are on the critical path for a handshake event. Therefore, it is extremely important to avoid unnecessary handshakes as much as possible, which Shasta/SMP attempts to achieve with the replication of the fine-grain tags.

Much like hardware implementations, upon access violations, Blizzard/S (Blizzard/SB) inserts the fault information into the fault queue. Upon a remote block fetch, a protocol handler first writes the data to memory and then upgrades the tag. Because protocol handlers exe-

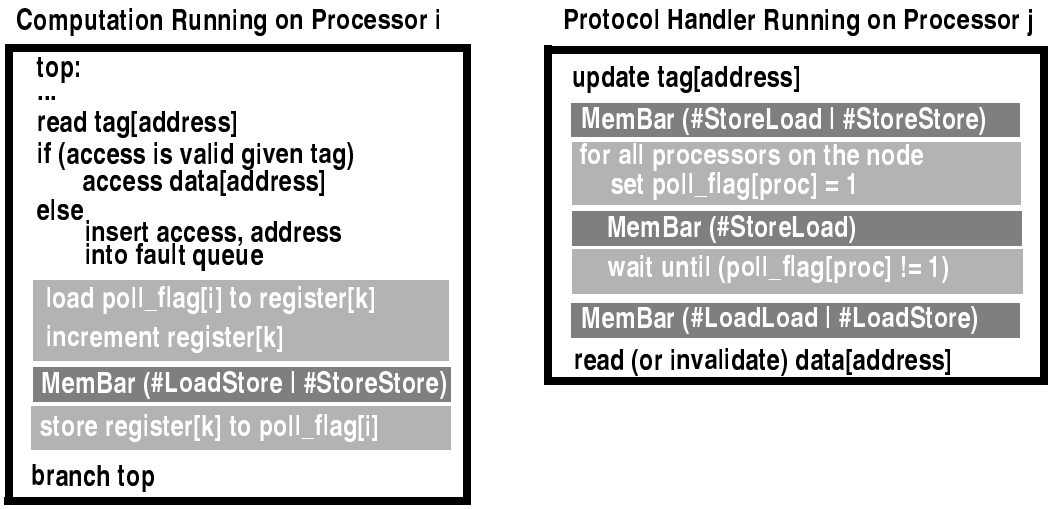

Figure 5-5. Software handshake with epochs.
The figure depicts the software handshake between the application (left) and protocol (right). Synchronization statements (shaded gray) in the application consist of three instructions to increment the value of the flag. The three instructions can be reduced to two if the epoch counter is kept in a global register. The protocol handler reads the flag value and waits until it changes. The SMP-node memory consistency model guarantees that the tag changes have propagated to all the processors. As in Figure 5-3 , memory barriers instructions (shaded dark gray) are required for correctness with weaker SMP consistency models.

cute to completion without interruptions, such an operation will appear to execute atomically with respect to the application.

Blizzard/E: Error-Correcting Code (ECC)

Blizzard/E uses ECC bits to identify invalid from read-only/read-write blocks, and distinguishes read-only from read-write blocks using virtual memory page table protection. Much like Blizzard/T, Blizzard/E uses custom bus error trap handlers to deliver access violations to user-level code. In Blizzard/E, much of the ECC manipulation is performed in the operating system using the interfaces described in Chapter 2. The atomicity of operations for setting bad ECC is guaranteed by suspending memory activity on all but one of the processors on the node. In every node, there is a master processor on the memory bus capable of masking the bus arbitration. When invalidating a block, the processor executing the protocol handler issues a system call. Unless this processor is the master processor, it sends an interprocessor interrupt to the master processor. The master processor reads the data into a user-accessible block buffer in memory, writes bad ECC to the memory block, and releases bus arbitration.

Unlike Blizzard/T, Blizzard/E uses page (rather than block) granularity protection to detect read/write access control by marking a page with at least one read-only block as a read-only page. Writes to writable blocks on a page containing a read-only block incur a page protection trap. The customized kernel trap handler also detects these special cases of access violation and executes the writes within the kernel trap handler directly. Writes to read-only blocks also trap, in which case the fault is delivered to the user-level.

Since writes to read-only pages are executed by the kernel trap handler after it accesses the fine-grain tags, the tag check and the write operation are not atomic. Therefore, handshakes are required when protocol handlers downgrade the block's tag. The scheme employed is similar to the handshake method in Blizzard/S, but with a slight difference. In particular, there is a single memory location for all processors that acts as a lock. The kernel attempts to set the lock before the write and the protocol handlers block while the lock is set. To protect against malicious processes, if a kernel fails to set the lock it returns control to the application and the access is retried. This scheme effectively serializes these writes through the kernel. However, these writes are much less frequent compared to instrumented memory operations in Blizzard/S. Therefore, it is preferable to reduce the handshake overhead on the protocol handlers than parallelize their execution.

Downgrading the tag from read-write to read-only may involve changing the page protection and consequently, a TLB shootdown when running on SMP nodes. Finally, Blizzard/E performs atomic upgrades of the tag while writing data much like Blizzard/T using uncached page mappings.

Blizzard/ES: A Hybrid of ECC and Software Tags

Blizzard/ES is an attempt to take advantage of features in both Blizzard/E and Blizzard/S. Blizzard/ES uses ECC to identify invalid memory blocks and software tags to distinguish read-write from read-only blocks. In comparison to Blizzard/S, Blizzard/ES eliminates instrumentation overhead on load operations altogether, but introduces the cost of maintaining ECC tags. Compared to Blizzard/E, Blizzard/ES eliminates the use of high-overhead page protection mechanisms at the cost of introducing instrumentation overhead for write operations. Much like Blizzard/SB, Blizzard/ESB implements the alternative form of application-protocol handshake.

5.4 Factors Affecting SMP-Node Performance

The performance of an FGDSM system on an SMP cluster depends on both application and system characteristics. Grouping processors is beneficial if the application sharing patterns favor fast (local) SMP hardware shared-memory mechanisms over high-latency (remote) FGDSM mechanisms. An SMP node also provides the opportunity for an idle processor to overlap running protocol handlers with computation on other processors. SMP nodes, however, introduce additional overheads, which may result in lower overall performance. In this section we identify the sources of overhead in SMP-node implementations, and quantify over-

head for common FGDSM operations. In Section 5.6, we use experimental measurements to evaluate the overall system performance.

We classify overhead into *SMP-correctness* and *SMP-contention overheads*. SMP-correctness overhead corresponds to the minimum overhead associated with SMP-node implementation of mechanisms in the absence of contention among multiple SMP processors to enforce correctness. We measure SMP-correctness overhead by comparing the cost of common FGDSM operations for SMP-node implementations running on uniprocessor nodes against the corresponding uniprocessor-node implementation. SMP-contention overhead refers to additional overhead due to contention among multiple SMP-node processors.

SMP-Correctness Overhead

SMP-correctness overhead corresponds to the minimum overhead to enforce correctness. Table 5.1 depicts the overhead of common FGDSM operations in various implementations of Blizzard without SMP-node support, and the additional overhead of supporting SMP nodes. Software SMP-node handshake incurs overhead of between 0 (for sentinel) and 2 (for table lookup) cycles upon tag check in Blizzard/S and Blizzard/ES, and 1 cycle upon loop-backedges in Blizzard/SB and Blizzard/ESB. Manipulating tags in ECC implementations may require a system call which incurs about 30 μs. SMP nodes may require an additional inter-processor interrupt (as described in Section 5.4) for an overhead of 30 μs if the system call originates from a processor incapable of masking memory bus arbitration. Minimum SMP-node overhead for tag manipulation in software implementations is simply the cost of hand-shake between the application and the protocol.

The table also presents remote miss times in Blizzard. The measurements correspond to minimum roundtrip miss times for a 128-block software DSM protocol between two machine nodes. The range of miss times corresponds to various types of remote miss, such as a read miss, a write miss, and an upgrade (write to a read-only block) miss. SMP nodes incur the additional overheads of passing information through a fault array and acquiring/relinquishing the protocol lock upon an access violation. In comparison, a uniprocessor-node implementation directly calls the appropriate protocol handler upon access violation and passes the fault information through processor registers. SMP nodes on the average increase roundtrip miss times by about 7-18% in Blizzard.

SMP-Contention Overhead

SMP nodes also incur SMP-contention overheads due to contention and sharing of resources among multiple processors. Contention for (local) memory accesses can lead to queuing delays on the memory bus. Similarly, contention for (remote) memory accesses can result in queuing delays for running protocol handlers. Applications not benefiting from SMP nodes exacerbate the demand for running protocol handlers by increasing the aggregate frequency of remote misses on a node. Allowing one processor to run protocol handlers on behalf of others may also result in one processor receiving a remote block requested by another processor. The

Table 5.1: Overheads in uniprocessor and SMP-node implementations of Blizzard.

The table presents the cost of FGDSM operations in uniprocessor-node (Base) implementations of Blizzard and the additional overheads in SMP-node (SMP Overhead) implementations. Tag check corresponds to the tag lookup overhead upon memory access. Backedge corresponds to the poll instrumentation overhead in every loop-backedge. Tag update corresponds to validating/invalidating, upgrading/downgrading of tags. Remote miss times correspond to roundtrip time from an access violation until resuming the access. The overheads vary depending on the type of the operation. For instance, software tag checks cost between 3 cycles for sentinel and 6 cycles for software table lookup instrumentation. The numbers indicate minimum overheads for the mechanisms. Cache effects can increase the overheads to higher values.

System	Tag Check (cycles)		Backedge (cycles)		Tag Update (μs)		Remote Miss (μs)	
	Base Cost	+ SMP Overhead	Base Cost	+ SMP Overhead	Base Cost	+ SMP Overhead	Base Cost	+ SMP Overhead
Blizzard/T	0	0	5	0	15-21	0	61-90	4-7
Blizzard/E	0	0	5	0	16-64	0-30	85-157	8-16
Blizzard/S	3,6	0,2	5	0	7-10	< 1	56-80	4-7
Blizzard/SB		0		1				
Blizzard/ES	0,6	0,2	5	0	14-58	0-30	77-146	9-13
Blizzard/ESB		0		1				

latter both pollutes the cache of the receiving processor and increases the roundtrip latency for the requesting processor by requiring an extra cache-to-cache transfer of data. Finally, for software tag implementations the handshake methods cause extra misses and therefore, memory bus traffic. By definition, this overhead can be measured when there is actual contention in the system. Moreover, we are interested in its effect on real applications and not some worst case bound. Therefore, any attempts to characterize this overhead should be postponed until Section 5.6.3, where application performance results are presented.

5.5 SMP Nodes and Protocol Traffic

Using SMP nodes affects the number of accesses to both local and remote memory. Changes in memory access patterns dictate the amount of contention to node's resources such as the

memory bus (due to local accesses), and FGDSM protocol processing (due to remote accesses).

Using SMP nodes aggregates the local accesses from neighboring (clustered) processors. Using SMP nodes also converts accesses among clustered processors from remote to local, while aggregating processors' accesses to remote nodes. Furthermore, because clustered processors share remote data, remote accesses by one processor may (implicitly) prefetch data subsequently referenced by another. The overall per-node number of resulting remote accesses in a clustered configuration depends on application sharing patterns.

Sharing patterns vary from strictly nearest-neighbor sharing in *em3d* and *tomcatv*, to mostly all-to-all sharing in *barnes*. Nearest-neighbor sharing result in the same aggregate number of per-node remote accesses in both clustered and uniprocessor-node configurations; on every node, both configurations result in exactly two immediate neighboring processors residing on remote nodes. In more complex sharing patterns, where processors do not favor sharing with their immediate neighbors, using SMP nodes increases the aggregate per-node remote accesses in the absence of implicit prefetching on shared remote data.

Depending on memory access patterns, an application primarily accesses either local or remote data. Using SMP nodes affects the performance of applications with dominant local memory accesses in three ways. First, clustered implementations at a minimum incur the SMP-correctness overhead of supporting SMP nodes. Second, an increase in local accesses can introduce queuing delays in the node's memory bus. Third, as described in Section 5.4, ad hoc scheduling of protocol handler execution among processors may result in adverse placement of data and an increase in the number of local accesses. Using SMP nodes affects the performance of applications with dominant remote memory access patterns in two ways. First, the aggregate per-node remote memory accesses change the demand for running protocol handlers. Second, SMP-node processors can overlap computation with protocol processing.

While the discussion so far has been focused on remote accesses examined from the application point of view, these accesses directly correlate with the protocol traffic generated by shared memory coherence protocols. In general, one can design a coherence protocol that generates an arbitrary amount of protocol traffic to satisfy remote accesses. However, with Stache, Blizzard's default shared memory protocol, things are fairly predictable. Stache is a standard four-hop directory based S-COMA protocol that reverts to a two-hop protocol when a page's home node is one of the sharers. If the application does not modify its accesses to remote data or the layout of the remote data in response to the change in the number of nodes, a link can be established between the application remote accesses and the generated protocol traffic.

To understand the implications of the increase in the clustering degree (the number of processors grouped into an SMP node) on the protocol traffic, we can classify the remote accesses in three categories: (i) accesses to data shared by future cluster neighbor processors, (ii) accesses to data that the future cluster neighbor processors access that they share with at

Table 5.2: Protocol traffic (in KB) per node as we change the clustering degree.

Measured is the traffic observed, direct is the traffic that was exchanged between future cluster neighbors in the 16x1 configuration, indirect all the other traffic saved due to the increase on the clustering degree.

	16x1	8x2			4x4		
Benchmark	**Measured**	**Measured**	**Direct**	**Indirect**	**Measured**	**Direct**	**Indirect**
em3d	11501	11501	11501	0	11501	34504	0
appbt	5273	4763	4608	1176	4789	13764	2541
barnes	1337	2207	177	290	2881	904	1566
tomcatv	352	331	372	1	288	1119	3
lu	1718	2293	568	576	3435	3417	22
water	384	523	109	137	551	438	549
em3d-cs	5557	5547	5546	21	5546	16651	32
appbt-cs	2475	3092	1857	0	4251	5647	3

least one non-neighbor processor, (iii) all the other remote accesses. The first class represents data directly shared by the cluster neighbors. Increasing the clustering degree allows the sharing to occur through the hardware shared memory and therefore, avoiding protocol traffic to move the data across nodes. We will refer to this as the *direct clustering benefit*. The second class represents data with more complicated access patterns. In this case, increasing the clustering degree acts as a form of implicit prefetching. We will refer to this as the *indirect clustering benefit*. The clustering benefit includes accesses to remote data by neighbor processors that reside in the same cache block even if they are not directly shared. Also, the reduction in the number of nodes can increase the indirect clustering benefit due to second order effects such as fewer invalidation messages for widely shared, fewer messages for global synchronization primitives like barriers or increased probability that one of the sharers will be the home node for a particular page.

For applications in which the amount of computation and data access patterns remains constant with an increase in the clustering degree, it is possible to compute the clustering benefit as follows. We measure the intranode traffic for each node, varying the clustering degree from one to four. Adding the traffic generated by the future cluster neighbors when the clustering degree is one, gives the protocol traffic per node if no clustering benefit had existed. The

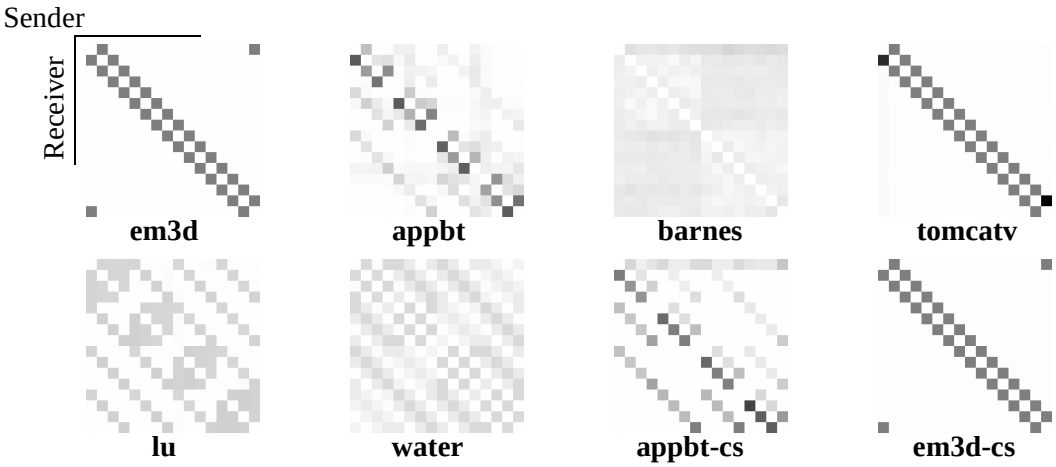

Figure 5-6. Application message traffic (sharing) patterns.

The figure illustrates application message traffic patterns running on our base system. Message traffic patterns in the base system to the first order approximate an application's sharing patterns. The shades of gray indicate message traffic intensity from senders (the x-axis) to the receivers (y-axis) quantitized to an 8-bit scale. Darker shades represent higher intensity. Top leftmost corner indicates traffic from node 0 to itself. For example, if 50% percent of the message generate by node i goes to node j and another 50% to node k then two dots both at gray level 128 will be depicted in coordinates (i, j) and (i, k).

direct benefit is computed by adding the traffic generated between future cluster neighbors in the base configuration. The indirect benefit is computed by subtracting the measured traffic in the clustered configurations and the direct benefit from the total traffic generated by the future neighbors in the base configuration. Table 5.2 presents the detailed results from measuring the direct and indirect clustering benefit for each application.

In the rest of this section, we will discuss six applications with different access patterns and communication requirements. For each application, we will examine the application and the data partitioning among processors as well as the application remote access patterns and how they correlate with the observed protocol traffic. Figure 5-6 illustrates the protocol traffic and therefore, the application sharing patterns on the base system for all applications. The sharing patterns also correspond to those in clustered configurations, because data partitioning in our applications is static.

EM3D. The graph characteristics determine the behavior on SMP nodes. In each iteration, every processor has to access *(PG/2D)* of the total graph nodes that belong to each of the processors within distance *D*. If this number is greater than one, it accesses some remote graph

nodes more than once per iteration. For D=1, accesses to remote graph nodes are restricted to the two immediate neighbor processors. As we have discussed already, this results in a constant amount of protocol traffic per node. Therefore, any performance effect due to SMP nodes must be because of factors other than the protocol traffic per node. For D>1, accesses to graph nodes are uniformly spread to the previous and next *D* processors. In this case, as we increase the clustering degree, the protocol traffic per node will increase. However, the probability that data accessed by one processor will also be accessed by one of its cluster neighbors increases. Therefore, the indirect cluster benefit will also increase. Since we have set *D=1*, for both the shared memory and the custom protocol versions, the number of data messages per node remains constant as the clustering degree increases.

Appbt. SMP-nodes eliminate remote data accesses along the faces of the subcubes shared between cluster neighbors. Since this is only one the three dimensions of sharing, the protocol traffic per node should increase with SMP nodes. This is what happens in the custom version. However, in the TSM version, accesses to the counters cause contention. Increasing the clustering degree, decreases the sharers for each cache block in the counter array and therefore, reduces the contention. This is manifested as indirect clustering benefit and reduced protocol traffic per node.

Barnes. This application does not exhibit any regular access patterns. Most of the communication happens during the tree build phase, which is not regular. Also, the data placement does not follow any regular patterns. Therefore, the home page for a body can be some other node that the ones that access it. The clustering benefit, direct and indirect, is not enough to avert an increase in the per node protocol traffic as the clustering degree increases.

Unlike Chapter 4, in these runs *barnes* uses MCS locks [MCS91] instead of the default Blizzard message locks, which have not designed to deal with multiprocessor nodes. The message locks are distributed among the nodes. Each lock or unlock request is implemented by sending an explicit message to the node that the lock resides. When multiprocessor nodes are used, this results in explicit messages exchanged even when both processors competing for the lock are on the same node. The Blizzard message locks were designed by the author because MCS locks did not perform well under high-contention. They have been subsequently evaluated by Kagi et al [KBG97] and their results suggest that the scheme satisfies its design goal being superior to any shared memory-based locks in the Blizzard environment. Presumably, an implementation of message locks aware of multiprocessor nodes can employ hierarchical schemes based on intranode local memory locks to coordinate the node processors and internode message locks to coordinate the nodes and efficiently implement message locks in an SMP environment.

Tomcatv. The input matrix is partitioned among the processors in contiguous rows. Sharing mainly occurs across the boundary rows. Therefore, it exhibits nearest neighbor communication.

LU. Cluster neighbors are located along the *y* dimension of the input matrix. This partition interacts strangely with SMP nodes. For 16 processors and a clustering degree of two, direct clustering benefit is realized among the intermediate neighbors in the second step of the algorithm and indirect clustering benefit is realized in the third step (see Chapter 4 for a description of the algorithm's steps). For a clustering degree of four, we observe only direct clustering benefit since all the processors along the *y* dimension belong to the same node. This sharing pattern suggests that the protocol traffic per node will increase with the clustering degree.

Water. For 16 processors, the partition algorithm creates 2x2x4 subcubes. Cluster neighbors are located along the *z* dimension. Sharing occurs among nodes that their subcubes touch at a base, edge or point. The 3D nature of the sharing pattern results in an increase of the protocol traffic with the clustering degree.

5.6 Performance Evaluation

In this section, we first present application speedups for our *base system,* a uniprocessor-node implementation of FGDSM without SMP-node overhead, and use it for performance comparisons against SMP-node implementations in the rest of the paper. We proceed by evaluating the impact of SMP-correctness overheads on application performance, and finally evaluate the performance of SMP nodes.

Table 5.3 presents the shared-memory applications used in this study, the problems they address, and their input parameters. *Em3d-cs* and *appbt-cs* [FLR+94] take advantage of the flexibility of software DSM and use customized protocols. In both cases, the applications bypass shared memory and use direct messaging to optimize communication.

5.6.1 Base System Performance

Table 5.4 presents application speedups running on the base systems. Speedups vary depending on applications' inherent parallelism, and their interaction with the FGDSM system (see Chapter 4). Applications with frequent access violations can benefit from efficient FGDSM implementations. Blizzard/T provides support for fine-grain sharing in hardware and always achieves the best speedups. Blizzard/S always incurs instrumentation overhead and favors applications with frequent access violations such as *em3d*. Access violations, in contrast, are very expensive in Blizzard/E, and hence it favors applications with less frequent access violations such as *tomcatv* and *water,* or no access violations such as *em3d-cs* and *appbt-cs*.

Because Blizzard/E uses page protection to distinguish read-only from read-write blocks, performance can suffer even in the absence of sharing if there are a large number of writes to read-only pages (i.e., pages with at least one read-only block) in the application. As an example, *lu* achieves reasonable speedups under Blizzard/T, but exhibits poor performance under

Table 5.3: Applications and input parameters.

Application	Problem	Input Data Set
em3d	3D electromagnetic wave propagation [CDG+93]	128K nodes, degree 6, 40% remote edges, distance span 1 (processors share edges only with their immediate neighbors), 100 iterations
appbt	3D implementation of CFD [BM95]	40x40x40 matrices, 4 iters
barnes	Barnes-Hut N-body simulation [WOT+95]	16K particles, 4 iters
tomcatv	Thompson's solver mesh generation [SPE90]	512x512 matrices, 100 iters
lu	Contiguous blocked dense LU factorization [WOT+95]	512x512 matrix, 16x16 blocks
water	Spatial water molecule force simulation [WOT+95]	4096 molecules

Blizzard/E. Blizzard/ES addresses this problem by performing tag lookups for stores in software and thereby obviating the need for read-only page protection. Blizzard/ES often either outperforms both Blizzard/E and Blizzard/S or performs close to the best of the two.

5.6.2 SMP-Correctness Overhead in SMP Nodes

SMP-correctness overhead represents minimum overhead incurred in order to support FGDSM on SMP nodes (Section 5.4). We measure SMP-correctness overhead by comparing Blizzard's SMP-node implementations running on uniprocessor nodes against our base systems. Figure 5-7 illustrates application execution times on SMP-node implementations normalized to those in the base system with the corresponding tag implementation. On the average SMP-correctness overhead is negligible (< 3%) in hardware tag implementations (Blizzard/T and Blizzard/E) and increases to up to 11% in the all-software tag implementation (Blizzard/S). Software tag implementations incur higher SMP-correctness overhead due to the application-protocol handshake.

Much like base system performance, SMP-correctness overhead also varies across applications. In Blizzard/T, applications with high sharing activity (such as *em3d*, *appbt*, *barnes*, and *em3d-cs)* exhibit higher SMP-correctness overhead. SMP-correctness overhead in Blizzard/S is higher for applications with frequent instrumented memory operations (such *barnes*, *lu*).

Table 5.4: Base system speedups for 16 uniprocessor nodes

Application speedups running on a uniprocessor-node (no SMP overhead) implementation of Blizzard with 16 nodes. The speedups are measured with respect to an optimized sequential (no FGDSM overhead) implementation of the applications running on one node of the machine.

Application	Blizzard/T	Blizzard/E	Blizzard/S	Blizzard/ES
em3d	3.8	2.2	3.7	2.7
appbt	6.2	2.8	4.5	4.8
barnes	6.1	3.8	5.3	4.7
tomcatv	10.6	8.8	6.9	9.0
lu	10.0	4.5	5.8	8.1
water	12.0	9.6	8.3	10.8
em3d-cs	17.2	16.4	12.2	15.7
appbt-cs	9.9	9.8	6.3	9.7

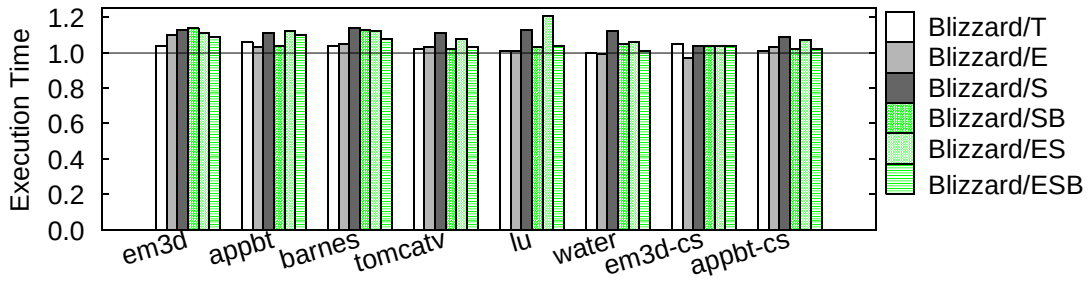

Figure 5-7. SMP-correctness overhead in Blizzard.
The figure presents application execution times under SMP-node implementations of Blizzard running on 16 uniprocessor nodes normalized to the base system (a uniprocessor-node implementation with 16 nodes).

The loop-backedge handshake on the average lowers SMP-correctness overhead in Blizzard/S (Blizzard/ES) by up to 4%.

5.6.3 Performance Impact of SMP Nodes

In this section, we investigate the impact of grouping processors into SMP node on application performance. We evaluate this impact by comparing Blizzard's SMP-node performance against that of our base system, while keeping the aggregate number of processors and amount of memory in the system constant. The performance impact of SMP nodes primarily depends on applications' memory access patterns (Section 5.4).

Figure 5-8 presents application execution times on 16 uniprocessor nodes, 8 dual-processor nodes, and 4 quad-processor nodes for all Blizzard implementations. The results are normalized to the corresponding base uniprocessor-node system. *Tomcatv, lu, water, em3d-cs* and *appbt-cs* are computation-intensive and primarily access local data, *em3d* and *appbt* are communication-intensive and frequently access remote data, and *barnes* accesses moderate amounts of remote data. *Em3d*, *tomcatv* and *em3d-cs* all exhibit nearest-neighbor sharing patterns (Figure 5-6). As such, SMP nodes do not affect the aggregate number of per-node remote accesses in these applications. *Appbt* uses shared-memory spin-locks and incurs frequent remote accesses on the critical path of execution. SMP nodes substantially reduce remote accesses on the critical path by converting spin-lock remote accesses to local accesses. Furthermore, SMP nodes slightly increase the aggregate number of per-node remote accesses in all the other applications (up to 50% in *barnes*).

Performance results indicate that SMP nodes both offer competitive performance and benefits from hardware support for atomicity. Dual-processor nodes perform very close to uniprocessor nodes for Blizzard/T and Blizzard/E and increase execution time on the average by 13% in Blizzard/S and 11% in Blizzard/SB. Quad-processor nodes also exhibit performance competitive to uniprocessor nodes, and are especially beneficial for Blizzard/E, converting high-overhead FGDSM operations (e.g., write to read-only pages) to fast SMP local accesses. This result corroborates previous findings for (high-overhead) DVSM implementations on SMP clusters [YKA96]. Quad-processor nodes also increase synchronization time in the loop-backedge handshake because a protocol handler must wait for three processors to reach a loop-backedge. This result indicates that the loop-backedge handshake may be suitable for small-scale SMP nodes while instrumented synchronization may be suitable for larger SMP nodes.

Em3d-cs consistently exhibits the largest performance degradation across all tag implementations. At the end of each iteration in *em3d-cs*, the system schedules one processor to receive all the incoming data. Because, the data are received in protocol processor's cache, subsequent accesses to the data by the consuming (i.e., computing) processor miss. Data belonging to other processors also pollute the protocol processor's cache. The combined effect significantly increases computation time in *em3d-cs* (> 50% for quad-processor nodes). This result suggests that systems should allow custom protocols to schedule protocol processing to a particular processor for effective cache utilization. The default scheduling policy favors request-response protocols in which the processor issuing the request resumes computation as soon as

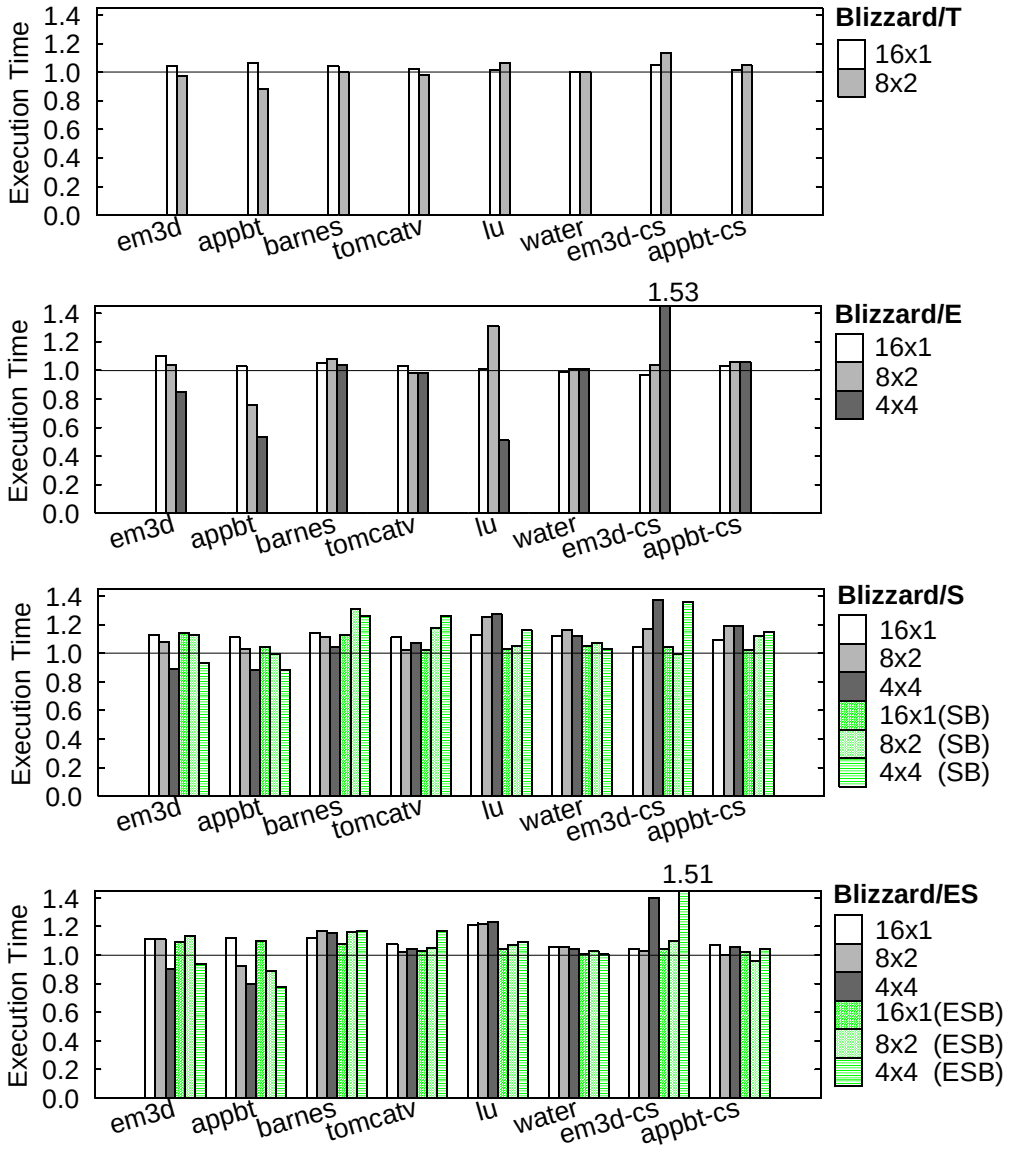

Figure 5-8. Performance of SMP Nodes in Blizzard.

The figure depicts application execution times under Blizzard/T, Blizzard/E, Blizzard/S, and Blizzard/ES for 16 uniprocessor nodes (16x1), 8 dual-processor nodes (8x2), and 4 quad-processor nodes (4x4) each normalized to the corresponding base system (a uniprocessor-node implementation with 16 nodes). Blizzard/T can only support up to two processors per node and does not have a quad-processor implementation. Blizzard/S and Blizzard/ES graphs also present execution time under the alternative loop backedge handshake implementations, Blizzard/SB and Blizzard/ESB.

the response arrives. Because responses arrive in the order in which the requests are sent, every processor processes the response to its own request.

Appbt and *em3d* significantly benefit from SMP nodes across tag implementations. In *appbt*, SMP nodes improve performance by reducing the aggregate per-node remote accesses. The impact on performance is more pronounced for tag implementations with high-overhead operations, such as Blizzard/E and Blizzard/ES. *Appbt* performs 90% and 27% better respectively on quad-processor than uniprocessor nodes. *Em3d* makes effective use of the processors' idle time to service remote requests. Because of nearest-neighbor sharing patterns, quad-processor nodes eliminate remote accesses for two processors. In every iteration, these processors complete computation more quickly than others and act as protocol processors on behalf of the node, overlapping protocol processing with computation and improving performance by up to 18%.

Lu exhibits irregular trends on Blizzard/E. Dual-processor nodes increase the frequency of write to read-only page operations. Because of serialization of these operations, performance suffers due to queueing delays. Quad-processor nodes significantly improve performance (96%) by converting these high-overhead FGDSM accesses to local SMP accesses.

SMP-node performance of software implementations depends heavily on the application-protocol handshake method. Looking at individual applications, it becomes an issue with relatively high number of access violations, as in *em3d* (on dual-processor nodes) and *barnes*, or when loop-backedges are very sparse, as in *tomcatv.* On the other hand, the loop-backedge method is consistently better for *lu* and *water* because of low frequency of protocol activity in these applications. Finally, the combined effect of high SMP-correctness overhead and an increase in protocol activity degrades SMP-node performance in *appbt-cs* by more than 10% with pure software implementations.

SMP-nodes do not necessarily have to achieve the best performance to prefer using them as building blocks for FGDSM systems. Rather, it may be sufficient if SMP nodes are *cost-effective,* i.e., if they achieve the lowest cost-performance [WH95] of all compared configurations. Using SMP nodes, many machine components such as the number of processors and memory requirements remain fixed across the platforms. Grouping processors to SMP nodes, reduces the number of required machine nodes and therefore, it results in a reduced number of motherboards, network interface cards, and network switches/routers.

It should be noted however, that although manufacturing cost of computer products is typically related to cost of components, cost from the perspective of a customer is related to price which is also dictated by market forces [HP90]. High-performance products, for instance, tend to target smaller markets and as such carry larger margins, charging higher premiums. In a related study [SFH+97] we found that today quad-processor servers command high price premiums. In contrast, dual-processor machines offer the lowest overall system cost. The main reason is that most system components used in single processor machines are designed to support up to two processors. Therefore, the cost of supporting dual processor systems is amor-

tized from single processor systems sales. Given the performance results in this section, unless the current trends change dual-processor nodes are likely to be more cost-effective as building blocks for FGDSM systems.

5.7 Related Work

Virtual-memory-based software distributed shared memory was originally proposed by Li and Hudak [LH89]. Many variations thereof have since emerged such as Munin [CBZ91], TreadMarks [KDCZ93], SoftFLASH [ENCH96], MGS [YKA96], and Cashmere [KHS+97]. These systems all use standard virtual address translation mechanisms to implement coherence across nodes, and rely on relaxed memory consistency models and careful program annotation to support fine-grain sharing. SoftFLASH, MGS, and Cashmere address SMP-node implementations of VM-based software DSM.

Fine-grain distributed shared memory has traditionally only been implemented in hardware. Blizzard/CM-5 is the first implementation of FGDSM either entirely in software, or with hardware assist [SFL+94]. Shasta [SGT96] is a successor of Blizzard and an all-software implementation of FGDSM. CRL [JKW95] is a region-based software FGDSM and requires program annotations for accessing regions. Neither Blizzard/CM-5 nor CRL have SMP-node implementations.

SMP-node implementations of Shasta [SGA97] have recently become available. Overall, Shasta/SMP appears to behave similar to Blizzard implementations performance-wise. This work builds upon theirs in several ways. First, the study evaluates how hardware (vs. software) implementations of sharing can help eliminate SMP-node overheads. Second, Blizzard implementations share the entire set of FGDSM resources among the SMP-node processors, allowing them to fully take advantage of SMP hardware coherence for sharing. Third, Blizzard implementations allow an idle processor on a node to run protocol handlers on behalf of the node, overlapping protocol processing on one processor with computation on others.

5.8 Conclusions

This chapter discusses Blizzard's support for SMP nodes. On a Blizzard node, processors share all of the FGDSM resources, such as the fine-grain tags, remote cache, home pages, directory, and fault and message queues. Sharing leads to a more effective use of resources and allows for a better utilization of processors by overlapping computation with protocol processing. Sharing resources, however, requires synchronization mechanisms to guarantee atomicity of accesses. The most frequently accessed resources in FGDSM systems are the memory and the fine-grain tags. We discussed low-cost software synchronization mechanisms to guarantee atomicity for these resources in the absence of hardware support.

The chapter evaluates the sources of overheads in SMP-node FGDSMs and classifies SMP overhead into *SMP-correctness* and *SMP-contention* overhead. SMP-correctness overhead corresponds to the minimum overhead associated with SMP-node implementation of mecha-

nisms to guarrantee correctness in the absence of contention among multiple SMP processors. Depending on the tag implementation, such SMP overhead exists in tag checks, tag changes, fault queue service. SMP-contention overhead refers to additional overhead associated with multiple processor on an SMP node contending for resources. SMP processors simultaneously contend for either access to local or remote memory. Contention of (local) accesses on the memory bus can lead to queuing delays. Similarly, multiple processors, depending on the application sharing patterns, can increase the demand for accessing remote memory and result in queuing delays for running protocol handlers.

The designs are evaluated using experimental results from both transparent shared memory applications and applications with customized coherence protocols. The experimental results on different clustering configurations show:

- SMP-nodes result in competitive performance while reducing hardware requirements for nodes, network interfaces and network switches. Prime factors to affect the performance of computation-intensive applications include: application sharing patterns, SMP overheads, memory bus delays, communication scheduling and implementation artifacts. Prime factors to affect the performance of communication-intensive applications include: application sharing patterns, overlap of communication and computation
- SMP-node implementations benefit from hardware support, which, in some cases, can completely eliminate the SMP overheads
- SMP nodes are especially beneficial from FGDSMs with high overhead operations.

Chapter 6

Address Translation Mechanisms in Network Interfaces

In this chapter, we diverge to some extent from the world of Blizzard and FGDSM systems to discuss an issue that applies more generally to network interfaces (NIs). In particular, we will focus on how to improve application message performance by eliminating all unnecessary copies (e.g., copying send data from user data structures directly to the NI). Throughout this chapter, the term “minimal messaging” will be used to distinguish it from many “zero-copy” protocols that employ user or system intermediate buffers.

Since FGDSM systems typically transfer data in small, cache-block sized quantities, the relative cost of copying is small, compared to the total cost of the messaging operation. However, application specific protocols can often change the picture [FLR+94] as they batch data transfers in larger messages. In general, however, this study extends beyond the domain of FGDMS systems. There are many networking applications that place a demand for high throughput and low latency on the network subsystem. On one hand, data intensive applications like multimedia depend on high throughput to stream large amounts of data through the network. On the other hand, client-server and parallel computing applications depend on low latency for fast response times. Network performance will become even more important as system area networks [Bel96] are used for clustered servers.

While the network hardware has been able to achieve high throughput and low latency, it has not proven easy to deliver this performance to the application. A key obstacle to reaching the hardware limits has been costs associated with message processing and message delivery within the host, especially when the operating system (OS) must be involved in every message transfer. For this reason, several research efforts have sought to provide protected user-level access to the network interface (NI), so the OS need not be invoked in the common case (user-

level messaging). Typically, the OS maps the device registers and/or memory into the user address space. Thereafter, the application can initiate message operations communicating directly with the device using loads and stores to send and receive messages. Examples of such designs include the Arizona Application Device Channels (ADCs) [DPD94], Cornell U-Net [vEBBV95], HP Hamlyn [Wil92], Princeton SHRIMP [BDFL96]. The result of such recent research has been commercial designs like Myricom Myrinet [BCF+95], Fore 200 [CMSB91], Mitsubishi DART [OZH+96].

With the OS removed from the critical path, the memory subsystem emerges as a major hurdle for delivering the network performance to the application. In the last ten years, memory speeds and memory bus bandwidth have failed to keep up with networks and the trend is likely to accelerate in the future [HP90,PD97]. This disparity has led some to argue that we are on the verge of a major paradigm shift that will transform the entire structure of information technology [Gil94]. Studies have shown that even today, network protocols spend a significant amount of time simply copying data [Ste94]. Therefore, many designs have attempted to avoid redundant copying at the application interface [DP93, RAC96, Wil92], the OS [kJC96], and the network interface [OZH+96, DWB+93, LC95, BCM94].

To push the envelope of possibilities, we should consider whether it is possible to efficiently implement messaging with no extra copying, where message data are only copied out of sender's data structures into the sender's NI and from the receiver's NI to the receiver's data structures. I call this property *minimal messaging* in place of the term "*zero-copy protocols*", which has been used inconsistently in the literature. Minimal messaging reduces the critical paths and decouples the CPU and the NI allowing the overlap of activities within the node. Moreover, it avoids second order effects like cache pollution due to messaging; bringing data into the CPU cache makes even less sense when the data originate or are destined for a device in the node (e.g., framebuffer or disk). Finally, it reduces the resource demands on the NI since data quickly move to the final destination. Thus, in many cases, minimal messaging leads to faster messaging.

For minimal messaging, the NI must examine the message contents, determine the data location and perform the transfer. The application accesses data using virtual addresses, which can be passed to the NI when the message operation is initiated. However, the NI is a device and thus it accesses memory using physical addresses. Therefore, the virtual address known by the application must be translated to a physical address usable by the NI; hence, an address translation mechanism is required.

Surprisingly, minimal messaging has often proven an elusive target. For example, in many designs that support minimal messaging, there are often restrictions placed to the size or location of the message buffers that can be directly accessed by the NI [BLA+94,GCP96,BJM+96,Hor95]. While the application can incorporate such message buffers within its data structures, in practice the complexity and cost of managing them can be significant and they may end up being used as intermediate buffers. We can trace the cause of such limitations to the address translation mechanism and its properties. Therefore, I argue

that to support minimal messaging, the NI must incorporate an address translation mechanism that (i) provides a flexible mechanism to be used by higher level abstractions, (ii) is able to cover all the user address space, (iii) is able to take advantage of the locality in the data source and destination to reduce the translation cost, and (iv) degrades gracefully when hardware or software limits are exceeded.

The address translation structures in NIs can be characterized by where the lookup is performed and where the misses are handled. For these questions, the answer can be either the NI or the CPU. Alternative design points differ in their requirements for OS support. Designs in which the NI handles misses, require extensive support by the OS (e.g., custom page tables). In contrast, handling misses in the CPU can be supported through standard kernel interfaces that wire pages in memory. Unfortunately, these interfaces have not been optimized for speed and therefore, such designs fail to degrade gracefully. Consequently, I discuss techniques to address this problem and offer graceful degradation in the absence of appropriate OS interfaces.

This study makes the following four contributions:

- It presents a classification of NI address translation mechanisms based on where the lookup and the miss handling are performed. It discusses alternative designs including a novel one that allows user software to load mappings via a device driver and therefore, it enables custom user-level control without sacrificing protection.
- It considers the OS support that alternative designs require and it proposes techniques to provide graceful degradation in the absence of appropriate OS interfaces including a novel one that gracefully degrades to single-copy when a mapping is not available and therefore, it makes fast miss handling less important.
- It provides performance data from simulations which demonstrate that the proposed techniques allow the designs to gracefully degrade. Moreover, the simulation results show that for the performance of the address translation structures more attention must be given to how misses are handled than to how the lookup is performed. For the lookup, software based schemes can give acceptable performance. For the miss handling, low overhead mechanisms are required or else extra care must be taken to avoid thrashing the translation structures.
- It demonstrates the feasibility of the approach by presenting experimental results from an implementation on real hardware within Blizzard’s framework where minimal messaging reduces latency for 2048-byte messages by up to 40%.

The remainder of the chapter is organized as follows.Section 6.1 elaborates on minimal messaging and the assumptions of this study. Section 6.2 identifies the address translation properties for minimal messaging. Section 6.3 analyzes the design space for address translation mechanisms and presents representative designs that span the entire design space. Section 6.4 discusses implications of minimal messaging for FGDSM in general and Blizzard in particular. Section 6.5 evaluates these designs in a simulation environment. and makes the feasibility case with an implementation within Blizzard. Section 6.6, presents related research

and commercial efforts and discusses their address translation mechanisms. Section 6.7 finishes the chapter with the conclusions of this work.

6.1 Minimal Messaging

In this section, I elaborate on minimal messaging and list the requirements that the NI must satisfy to be able to support minimal messaging. Furthermore, I state the assumptions that are implicit in this study and sketch the simplified system model that is used throughout the rest of the chapter.

For minimal messaging, the NI must examine the message contents, determine the data location and perform the transfer (Figure 6-1). The first step requires the NI to have processing capabilities. The second step requires a translation mechanism from virtual addresses to physical addresses. The application knows the data location by its virtual address. It can be passed to the NI when the message operation is initiated. However, the NI is a device, commonly attached to the I/O bus. It can access memory using physical addresses. This dictates that the virtual address known by the application must be translated to a physical address usable by the NI, in a protected manner. For the third step, the NI must be able to access the data in physical memory without requiring kernel involvement to flush the CPU cache before or after the DMA operation (processor coherent DMA). This is supported in modern I/O architectures [Gro95,Mic91].

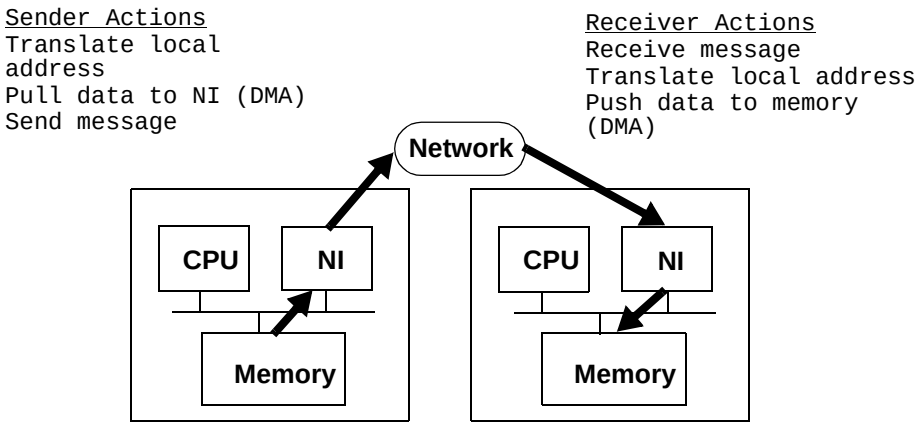

Figure 6-1. Minimal messaging event sequence.

The figure illustrates the event sequence on the sender and the receiver with mir

Since physical addresses must be used to read data from the sender's memory and write data into the receiver's memory, the physical address must be available to the NI before the data movement can take place. I will assume that this is done with two address translations: at the sender and at the receiver using user's virtual addresses (Figure 6-1). Alternatively the sender could perform the receiver's address translation and send messages with destination physical addresses [BLA+94]. I will do not consider this case further, because non-local knowledge of

physical addresses makes paging, fault isolation and security containment much more difficult.

The assumption throughout this study is that the NI address translation mechanism operates on the virtual addresses as known by the application. In a general purpose OS, this is not sufficient since all processes in a node view the same address space. Therefore, the mechanism must be extended for multiple senders and receivers in one node. A straightforward way is to define message segments as areas where messages can be send or delivered. Applications can create such segments to export a region of their address space. Thereafter, the equivalent of the virtual address is the pair of <segment id, segment offset>. Subtracting the segment base from an application virtual address is sufficient to calculate the segment offset for any virtual address within the message segment. A protection mechanism can grant access for a particular process to send or receive messages to a specific message segment. Thereafter, the NI must enforce the access rights when it sends or receives messages. Moreover, extra care must be taken to break the remote connection when a process is destroyed. An equivalent model is described in detail in Berkeley's Active Message specification [MC95] or Intel's Virtual Interface Architecture [DR97].

6.2 Address Translation Properties For Minimal Messaging

In this section, I present the properties that the address translation mechanism should satisfy and I argue why I consider them desirable for minimal messaging. First, NI address translation mechanisms must be incorporated within an abstraction that can support minimal messaging. The first two properties, application interface requirements we often see violated in existing designs, belong in this category. Second, the performance characteristics of the address translation mechanisms must favor minimal messaging. The next two properties, performance requirements that allow minimal messaging to provide better performance than single-copy messaging, belong in this category.

(A) Provide a consistent, flexible interface to higher level abstractions. The address translation mechanism should allow the cut-through semantics of minimal messaging to be exposed to the application through the abstraction layers. In most cases, applications access the network through a layer of messaging abstractions. We can distinguish between low-level network access models and high-level user messaging models. Network access models such as ADCs [DPD94], U-Net [vEBBV95], Active Messages [vECGS92], Fast Messages [PLC95], provide protected user access to the NI and serve as a consistent low-level model across NIs. Applications can use them to access the network but likely they will prefer higher level messaging models such as Fbufs [DP93], MPI [For94] or TCP/IP.

Minimal messaging, by definition, provides a path for the message data through the abstraction layers to the application data structures. At the lowest level, incoming or outgoing messages should point to application data structures. Until message operations complete, these structures are shared between the application and the NI (*shared semantics*). In abstraction layers that offer *copy semantics*, in which outgoing or incoming messages contain copies of

the application data, it is difficult to fully support minimal messaging. In general, a change in semantics introduces extra overheads [BS96]. Therefore, it is important that the lowest abstraction layer in the NI architecture to offer appropriate semantics or else implementations of all abstraction layers will suffer from the mismatch in semantics.

As a negative example of inflexible low-level interfaces, consider ADCs, which have been designed to optimize stream traffic. On the sender, the application enqueues the data addresses for outgoing messages. On the receiver, the data end up in incoming buffers allocated out of a queue of free buffers in the application address space. An address translation mechanism can be used to map the application addresses to physical addresses [DAPP93] but it is not sufficient to fully support minimal messaging because the abstraction does allow the NI to move incoming data directly to user data structures. Except for limited cases where the application knows the data destination for the next incoming message before the message arrives, an extra copy on the receiver is necessary.

In the designs I present in Section 6.3, I avoid such limitations in the application interface. The only requirement in the application interface is that the remote virtual address should be specified when the message is injected into the network. If this address is not known, the messaging library resorts to a single-copy approach. Alternatively, I could have used more aggressive application interfaces that allow the remote virtual address to become known just before the data are moved to the receiver's memory [Osb94,RAC96].

(B) Cover all the user address space. If the address translation mechanism is limited in its reach or it is expensive to change which pages it maps, the available space may end up being used as intermediate buffers in a single-copy approach. While it is possible for the application to incorporate these message buffers within the application data structures, in practice the complexity and overhead in managing them can be significant, depending on the application characteristics. For example, if the application wants to use the mechanism for a memory region of size greater than the mechanism's reach, it has to make explicit calls to change the installed mappings as it sends or receives different portions of the region. This limits the portability of the applications since it exposes them to an implementation constraint. More importantly, it becomes impractical to use the mechanism for applications without a prior knowledge of the messaging pattern or without a message exchange to agree on the translations before the data transfer. Finally, it becomes difficult for higher level messaging models to expose such constraints to the application cleanly without violating the first requirement.

The reason for the limited reach in NI address translation mechanisms is often the absence of *dynamic miss handling*. If a translation is not available, no mechanisms exist to install the translation (e.g., loading the translation from device page tables or notifying the OS to do it). Designs that lack this feature are limited by the size of the NI translation structures. The designs of Section 6.3,avoid such limitations because the translation structures are treated as caches that dynamically respond to the application requirements.

(C) Take advantage of locality. In any translation scheme we would like to take advantage of potential locality by keeping recent mappings handy for future uses. Applications exhibit temporal and spatial locality at various levels, which include their network behavior [Mog91]. This locality will be reflected in the application data addresses from which the applications send or receive messages. The address translation mechanism should take advantage of this behavior when it exists.

(D) Degrade gracefully when system limits are exceeded. Locality by itself should not be the only mechanism which ensures good performance. First, the NI translation structures cannot fully describe the address space. In the unlikely case, they are large enough to contain as much information as the kernel structures (i.e., page tables), performance considerations make it too expensive to maintain a copy of the kernel structures on the NI. Therefore, we shall have to deal with misses when we exceed the capacity of the NI translation structures. Second, we should not degrade the performance of transfers that do not exhibit locality, such as one-time bulk data transfers. Thus, we should strive for a translation mechanism that allows minimal messaging to be at least as good as the single-copy approach when its limits are exceeded. As we shall see in Section 6.5.2, minimal messaging can easily result in worse performance than single-copy messaging, if the design requirement for graceful degradation is overlooked.

6.3 Address Translation Implementation Alternatives

This section presents a classification of the NI address translation mechanisms according to where the lookup and the miss handling are performed. Then, I examine the points in this design space and discuss the requirements of alternative designs for OS support. The discussion points to the central role of the OS in providing the interfaces required to achieve graceful degradation. Accordingly, I finish this section discussing three techniques to make this property hold in the absence of appropriate OS interfaces.

NI address translation structures can be viewed as caches that provide physical addresses for data sources and destinations. Two key questions in a cache design are how to do the lookup and how to service misses. There are two places, the NI and the CPU, where the lookup can be done. When the lookup is performed on the CPU, misses will be handled there. When the lookup is performed on the NI, there are two choices on where to handle misses. The first is for the NI to directly access device page tables (prepared for the NI) in main memory and handle its own misses. The second is for the NI to interrupt the CPU and ask it to service the miss.

Designs that perform the lookup and the miss handling in the NI correspond to network coprocessors or network microcontrollers [BCM94,Int93]. Designs that perform the lookup in the NI and the miss handling in the CPU, correspond to software TLBs or custom hardware finite state machines [HGDG94,OZH+96]. This classification reveals another interesting design point in which both the lookup and the miss handling are performed on the CPU through an interface that allows user-level software to control the mappings that are installed in the NI translation structures. Table 6.1 shows the design space and places representative designs within it.

Table 6.1: Classification of Address Translation Mechanisms

Lookup		Miss Service	
		NI	**CPU**
NI	**Hardware Structures**	Network Coprocessors (Section 6.3.1)	Custom Designs (Section 6.3.2)
	Software Structures	Network Microcontrollers (Section 6.3.1)	Software TLBs (Section 6.3.2)
CPU		-	User-controlled mappings (Section 6.3.3)

6.3.1 NI Lookup -- NI Miss Service

We can build a flexible NI that handles its own misses. The address translation mechanisms in this design follow the philosophy of similar mechanisms designed for processors. Modern operating systems maintain three levels of processor translations. First, *translation lookaside buffers (TLBs)* make mappings available to the CPU(s). Second, processor page tables are maintained in main memory from which CPU(s) quickly load mappings for pages resident in main memory. Third, a complete description of a process address space, including pages that have been swapped out or not accessed yet, is maintained in internal kernel data structures [Vah96]. The OS defines public kernel interfaces to access and modify mappings in all these levels. In addition, it maintains consistency among the different translation levels. For example, when a page is swapped out to secondary storage, any mappings for that page must be flushed throughout the system.

In this design (Figure 6-2) the OS views the NI as a processor. Address translation structures on the NI correspond to CPU TLBs. The NI searches for mappings when it sends or receives a message in order to access application data. If a translation is not available in the NI translation structures, the NI accesses device page tables that the kernel maintains in main memory. This should be an operation of the same order as a CPU TLB miss (~ few hundred cycles). If the page has not been accessed before on the node or it has swapped out, the kernel in the host CPU should be invoked to take care of the miss. In the general case, it is not realistic to expect that the NI is able to execute kernel code. This requires a network coprocessor of the same architecture as the host CPU with the ability to access kernel data structures efficiently and communicate with devices (e.g., disks), which may be impossible for a device on the I/O bus. I do not consider such misses further because both in single-copy and minimal messaging the limiting factor is how fast the kernel can allocate new pages or swap in old pages from secondary storage.

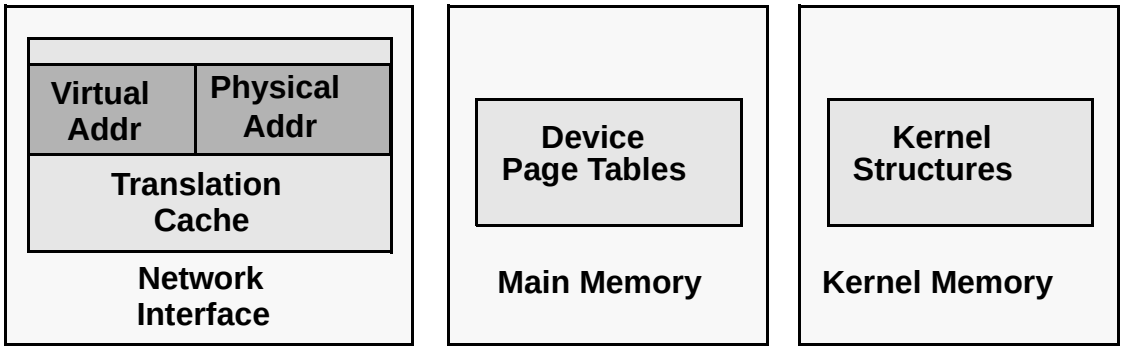

Figure 6-2. NI Lookup -- NI Miss Service.

A translation cache is physically present on the NI. If a translation is not available in this cache, the NI looks up device page tables that the kernel maintains in main memory. For misses on pages that have not been accessed before on the node or that have been swapped out, kernel code must consult the OS structures that describe the application address space.

We can implement NI translation structures in software, similar to the software TLBs proposed for FLASH [HGDG94]. To implement software structures, we need an NI microcontroller that it is flexible enough to synchronize with the node's CPU to access its own page tables in main memory. Such structures have small associativity and many entries. The lookup overhead is directly proportional to the number of entries in the cache set that we must examine sequentially to find a match. Alternatively, we can consider hardware support for the lookup as in designs with a network coprocessors that include their own memory management unit and address translation hardware (TLBs) [BCM94,LC95]. Such hardware structures should have high associativity with relative few entries (~ tens) and zero lookup overhead.

When a mapping is invalidated (paging activity, process termination) throughout the system, the host CPU must flush the entry out of the NI page tables, This is an operation that it is similar to TLB invalidations in multiprocessor systems. This is more complicated than in the case of a processor TLB because processor translation apply on memory accesses that are atomic with respect to other system events. NI translations, however, must be valid for the entire duration of the data transfer. This requires either that the active transfers aborted or the kernel is made to avoid invalidations for pages with active transfers.

The main disadvantage of this design is that it requires significant OS modifications since the OS must treat the NI as a processor. Commodity operating systems have not been designed to support page tables for arbitrary devices and they do not offer public kernel interfaces that provide this functionality. Even worse, this functionality cannot be implemented by standard device drivers using unsupported kernel interfaces because the virtual memory subsystem is at

the heart of an OS and it cannot be easily modified without kernel rewriting. Therefore, including the appropriate support in commodity operating systems requires significant commitment from the OS groups for a specific platform. For example, in Solaris 2.4, the only device (other than processors) for which the kernel supports page tables in the Sun-4M architecture [Mic91], is the standard SX graphics controller [Mic94]. The code is deep inside the virtual memory subsystem and no public kernel interfaces exist to support this functionality for other devices or even other graphics controllers.

6.3.2 NI Lookup -- CPU Miss Service

To overcome the lack of OS interfaces that support device page tables, we can handle misses in the host CPU within the device driver's interrupt routine using standard kernel interfaces (Figure 6-3). The key characteristic of this design is that it supports only the NI translation structures and not special device tables in main memory. NIs with microcontrollers can implement the lookup with software TLBs [HGDG94]. Some designs however, include hardware support for the lookup [OZH+96] in the form of custom finite state machines for message processing.

Whenever a miss occurs, the device triggers an interrupt invoking the device driver's interrupt handler, which handles the misses. Using standard kernel interfaces, pages are wired down when their mappings enter the translation cache and are unwired whenever they leave it. The OS must know the reason for wiring the page down (i.e., incoming or outgoing message) to properly maintain the dirty and reference bits. Because of this, if a page is wired for an outgoing message, a second fault must be forwarded to the host when a translation is required for an incoming message.

In this design, the OS treats NI translations as device translations. However, the way that the NI uses the translations does not match the way that the OS expects device translations to be used. Device translations are explicitly requested on an operation basis (e.g., when transferring data from disk to memory). The kernel defines an interface to wire pages in physical memory before the transfer and unwire it when it is completed. Since the OS sets up the translation, it is aware of incoming or outgoing transfers and it can maintain dirty and reference bits for these pages. In our case however, the NI keeps pages wired as long as they are installed in the translation cache, potentially for a long time. Since the OS assumptions about the duration that the pages remain wired do not match reality, we have an OS integration problem. Effectively, wired pages are no longer managed by the OS and this may result in underutilization of the physical memory. It is possible that these pages remain wired in memory even if the translation is not useful for communication anymore. If replacements due to conflicts are not enough to force old mappings out of the cache, we must periodically flush it.

In addition, the kernel interfaces involved for wiring and unwiring pages are not particularly fast. These calls must traverse layers of kernel software to perform the operation (~ few thousand cycles). If the translation structures cannot hold all of the translations used by the application, miss processing within the device driver is in the critical path. Therefore, the ability of

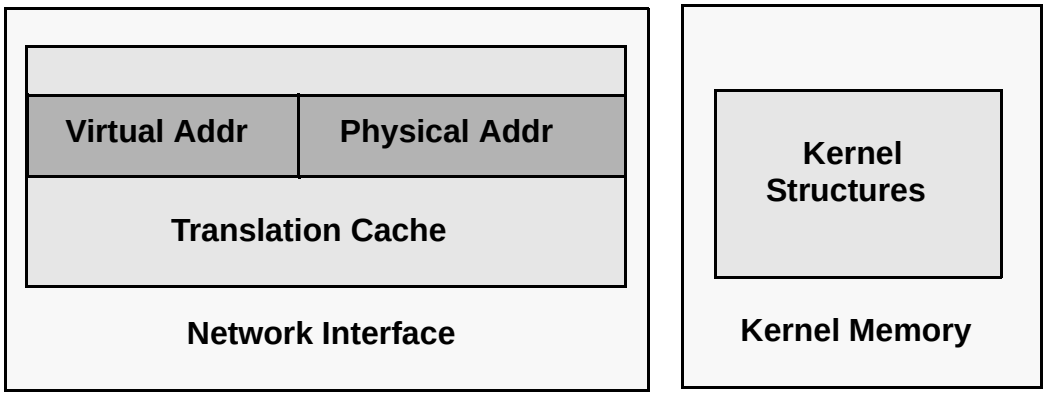

Figure 6-3. NI Lookup -- CPU Miss Service.

A translation cache is present on the NI. The NI device driver uses the processor page tables (memory resident pages) or the kernel's address space structures to load mappings in the NI translation cache.

the design to gracefully degrade is questionable once system limits have been exceeded and the translation structures start thrashing.

6.3.3 CPU Lookup -- CPU Miss Service

Traditional NIs often support minimal messaging with the kernel [kJC96,BS96,TLL94] supplying physical addresses to the NI. Involving the kernel in every message operation is unacceptable for user-level messaging. Therefore, I have devised a novel mechanism that allows us to do the lookup in user level on the host CPU (Figure 6-4). The key idea is to provide a protected interface through which user code can manipulate the contents of the NI translation structures. This enables custom user-level control without sacrificing protection.

In this design, there is a table on the NI that holds physical addresses. The user code cannot directly modify the contents of that table. However, it can request the device driver to install the physical address of a page in the user address space at a specific table index. The OS checks the validity of the user request, wires the page in memory, unwires the page previously installed there and stores the physical address in the requested table entry. Thereafter, the user code can pass to the NI the table index. The physical address stored in the table can be used as the data source of an outgoing message or the data destination of an incoming message.

For outgoing messages, the translation is done by the CPU before the message is processed by the NI. If the size of the table is greater than the maximum number of outstanding messages and we are using a *least-recently-used* (LRU) algorithm to choose replacement candidates, it is guaranteed that the translation will remain valid until the message is processed by

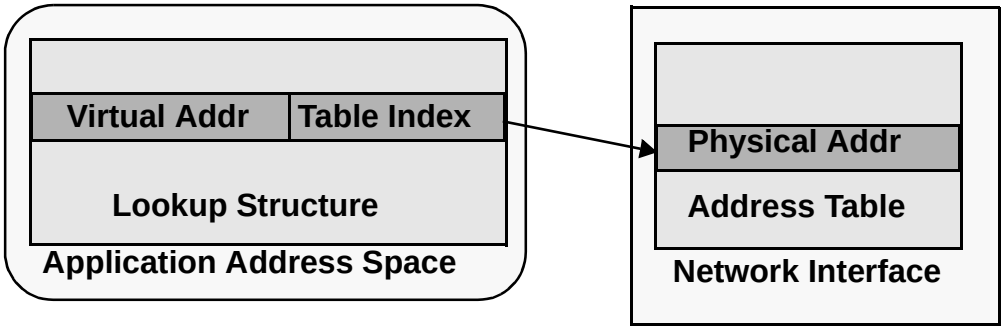

Figure 6-4. CPU Lookup -- CPU Miss Service.

Data structures in the application address space are used for the lookup. The device driver installs physical addresses in the translation table per user request using the processor page tables (memory resident pages) or the kernel's address space structures.

the NI. For incoming messages, the translation is performed after the CPU examines the header of the incoming message. This results in a distinct disadvantage of this design over the design in Section 6.3.2. On the receiver, the CPU is still on the critical path since we cannot know the destination address for the incoming message until after the CPU examines the message and moreover, we have to pay the notification penalty for the message arrival. We can address this shortcoming by specifying the destination virtual address when the message is sent as the index in the receiver's translation table. This scheme, suggested by David Wood [Woo97], requires that each node to maintain a local copy of the receiver's translation table for all potential destination nodes. Accordingly, the nodes must collaborate to keep these copies consistent and inform each other of changes in the translation maps. Before a table entry can be locally replaced, it must first be replaced in all the potential sender nodes. The effectiveness of this scheme depends on the overhead of the notification cost vs. the overhead in maintaining the translation tables consistent and the application messaging activity. This evaluation is beyond the scope of this study.

This design has two advantages over the one in Section 6.3.2. First, the policies are set entirely at user level, so any user-level application knowledge can be readily incorporated. Second, it can support sophisticated algorithms to manage the translation cache since the host CPU typically is more powerful and flexible than any NI logic or NI processor. For example, in our user-level messaging library, we use this processing power to maintain a three-level page table in user software. On the bottom level, we store records to keep track of whether or not a translation for that page is installed in the table. Furthermore, the pages with active translations are also installed in a double-linked list that implements the LRU policy.

The design also shares the same problems with the one in Section 6.3.2. First, as we have discussed, the OS integration is problematic since standard kernel interfaces for wiring pages have not designed for the way we use them. Second, the design fails to degrade gracefully due

to the overhead in standard kernel interfaces for wiring pages. If the physical address table cannot hold all of the translations used by the application, the device driver system call to install new mappings is in the critical path for all messaging operations once thrashing starts. The presence of an interface for user control of the replacement policy partially offsets this problem since it gives some user control to avoid thrashing. However, it is not desirable to expose this interface all the way to the application since it is implementation specific.

6.3.4 CPU Miss Service Optimizations

The discussion thus far favors designs where misses are handled by the NI. Since this depends on how the OS views the NI, lack of OS support can pose a significant obstacle. To overcome this problem, we can handle misses on the CPU. In this case, we can implement all the functionality in a device driver with standard kernel interfaces. However, as we have discussed, this approach fails to degrade gracefully once the translation structures start trashing. Therefore, we propose some techniques to avoid this worst case scenario.

Reduce the miss rate. We can reduce the miss rate in cache designs with better replacement policies. However, no policy can avoid thrashing when the application requirements exceed the size of the translation structures. Alternatively, we can increase the number of translation entries, either by increasing the amount of memory in the NI board or by using a second-level translation cache in main memory. In designs with second-level translation caches, unlike the designs of Section 6.3.1, the OS still treats the NIs as devices. Therefore, pages with translations in the NI translation structures must be wired using standard kernel interfaces.

Increasing the number of entries in the translation structures makes the problem of poor memory utilization more severe. Because there are many entries, flushing the entire cache may become too expensive to do indiscriminately. Therefore, we need to replicate the kernel's paging algorithms for the wired pages in the NI and its device driver. In particular, we must maintain dirty and reference bits and periodically sweep unused pages by flushing their translations from the NI translation structures.

While larger translation structures increase the reach of the NI address translation mechanism, the fundamental problem of graceful degradation has not been addressed. If the size of the application working set is larger than the size of the translation structures thrashing still occurs. Therefore, we do not consider it further.

Reduce the miss penalty. Despite the fact that kernel interfaces for wiring pages are not very fast, we argue that the operation is not logically complicated. The kernel must translate a virtual address to a physical address and keep track that a translation is being used by the NI. We can therefore, devise a specialized, fast kernel interface as follows. In the common case, we can use the CPU translation hardware by temporarily switching contexts to the process that the virtual address belongs to and then probe the memory management unit (MMU) to get the translation from the hardware page tables. Then we can update the NI translation cache and set a bit in a kernel structure to ensure that the page is wired. The paging algorithm must

be modified to take these bits into account. These operations can be performed in the kernel trap handler fast (~few thousand cycles). Using similar techniques, we have been able to reduce the roundtrip time for synchronous traps on 66 MHz HyperSparcs, from 101 μsecs with the standard Solaris 2.4 signal interface to 5 μsecs with optimized kernel interfaces [SFH+96]. Other researchers have reported similar results [RFW93,TL94b].

However, specialized fast interfaces are not viable in the long run. First, we violate kernel structuring principles. Device specific support must be implemented at the lowest kernel levels where no public interfaces exist to support this functionality. The exact details of the handler depend on the processor and system architecture as well as the OS version. A different handler is most likely required for every combination of device, OS version, processor implementation, and system architecture (if the OS natively supports such an interface, its development becomes more manageable). Second, it will become more difficult to maintain good performance with this technique in the long run. As CPUs get more complicated (e.g., speculative execution), larger amounts of CPU state need to be flushed on a trap, thereby increasing the trap overhead. Nevertheless, the method is a good way to get acceptable performance in prototype implementations.

Tolerate the miss penalty. For graceful degradation, it is sufficient to be able to fall back in the performance of the single-copy approach when the translation cache exhibits thrashing. This can be achieved by defining the single-copy as a fast default fallback path for the message to follow while handling translation misses is postponed. This strategy is most effective when the miss detection and the decision to use the fallback path occur at the same place, that is with host lookup for outgoing messages and with NI lookup for incoming messages.

With this technique, the NI may or may not move the data to the final destination. If it does not, some code running within the messaging library emulates the correct behavior by copying the data to the destination. This design moves the miss service off the critical path. Minimal messaging becomes an optimization that the system uses when possible. We can further enhance the mechanism's flexibility by allowing the user code to initiate miss service through an explicit request at appropriate times.

By itself, the existence of the single-copy fallback path is not enough to track the performance of the single-copy mechanism. We postpone servicing miss requests, but we still have to do them. However, the existence of the single-copy fallback path allows us to ignore those requests. Therefore, we can achieve graceful degradation if we control the rate of miss processing. This can be done using internal knowledge (e.g., avoid it when we know we will exceed the capacity of the translation structures), directly exposing it to the higher layers, or providing worst case bounds. We have implemented a worst case bound as follows: miss processing should not degrade the performance of the worst case by more than a predefined amount over the single-copy method. A simple hysteresis mechanism that counts the message bytes transferred and only services misses once every *n* bytes, approximates this constraint. Effectively, we sample the address stream once for every *n* data bytes transferred. In other

words, we trade the ability to take advantage of locality present in highly variable streams for worst case bounds.

6.4 FGDSM Systems & Minimal Messaging

I have incorporated minimal messaging in Blizzard to demonstrate the feasibility of the approach on real hardware. In this section, I describe the issues involved in supporting minimal messaging within existing messaging interfaces in general, and FGDSM systems in particular. Moreover, it serves as an example of including minimal support in other messaging interfaces. The section first discusses general issues regarding the introduction of minimal messaging in an existing messaging interface and subsequently focuses on the interaction of minimal messaging with fine-grain access control.

Blizzard supports Tempest's [Rei94] messaging interface. The Tempest messaging interface implements an active message model where upon a message arrival a message handler is invoked. It supports small and large messages. Small messages transfer few words. Typically, these messages are implemented by including the word data in the message header [SFL+94, SFH+96]. Therefore, minimal messaging is not applicable to these messages. Large messages transfer either word or cache-block aligned regions. The former are used for general messaging operations while the latter are optimized for transferring cache blocks. Moreover, cache-aligned messages facilitate the development of shared memory coherence protocols by supporting calls which can atomically change a block's protection and send the block data.

Originally, Tempest supported receiver-based addressing, in which the handler invoked at the receiver end upon message arrival could specify the destination for the incoming message data using a special Tempest primitive. Obviously, this means that the message data cannot to be transferred to their final destination in the receiver before the handler has been invoked. In anticipation of supporting minimal messaging, the interface has been extended to support sender-based addressing. A second set of send primitives allows the remote virtual address in the destination to be specified when the message is sent. When the handler for messages sent through these primitives is invoked, the data have already been copied to the final destination. Depending on the implementation of minimal messaging, this may have happened by the host processor within the messaging subsystem (e.g., when using the single-copy fallback). In this way, the implementation complexity of minimal messaging is not exposed to higher levels.

The original Tempest interface specifies primitives with synchronous send semantics. In other words, the assumption is that after the operation was invoked the data area could be modified without consequences. To support such semantics with minimal messaging, the send primitive does not return until after the data have been pulled down to the NI. While this allows to support minimal messaging within the old interface, it also limits performance. In fact, the processor and the NI are tied even closer than using intermediate buffers. With minimal messaging, the maximum rate with which messages are generated cannot be greater than the maximum rate that the NI can send them while intermediate buffers allow temporary traffic spikes to be accommodated without slowing down the host processor. To alleviate this

problem, a new set of messaging primitives with asynchronous semantics has been introduced. These calls schedule messages and return immediately with a token. It is the responsibility of the user code to check that the messaging operation has been completed by invoking with this token an appropriate query primitive that informs the caller about the status of that message.

The interactions of fine grain access control with minimal messaging introduce additional complications. In particular, two conditions must be met. First, on the sender, the tag change must allow the NI to access the data in the original place while they remain inaccessible by the processor(s). Second, on the receiver, the NI must be able to deposit the data without inadvertently upgrading the tags to a greater level than intended. The difficulty of meeting these conditions under different tag implementations depends on the details of the tag implementation.

In Blizzard/S, due to the use of sentinels, which require that the memory is overwritten with the sentinel values when a block is invalidated, the first condition is not satisfied. The block cannot be invalidated before the data are pulled down to the NI. Therefore, either the NI or the processor must complete the invalidation by writing the sentinel values in memory after the messaging operation.

In Blizzard/E and Blizzard/ES, a similar problem also exists. However, in this case it is more difficult to address it since the ECC invalidation operation must be performed inside the kernel. Given that the kernel has to copy the data out of the original location and therefore, bring them in the processor cache, there is not much incentive for minimal messaging. Blizzard/E also fail to satisfy the second condition when the a readonly block is deposited directly in the receiver's memory. If the page protection is not readonly, nothing protects the block from store operations from the node's processors. A possible solution is to maintain the page protection on the NI translation cache. Then, we can treat as a miss incoming readonly blocks to readwrite pages.

Blizzard/T allows to alleviate such problems because we can selectively allow access to a portion of memory depending on who is doing the access. Vortex does not enforce fine-grain semantics on NI accesses and therefore, satisfies both conditions.

Although minimal messaging support for cache-block aligned messaging primitives, as described above, had been implemented at some point in Blizzard, it was later rolled out due to the effort required to maintain distinct versions of the messaging primitives for every tag implementation. Currently, Blizzard emulates message primitives with sender-based addressing without using minimal messaging. On the sender, data are explicitly copied by the processor. On the receiver, the data transfer and tag change are performed by the processor before the message handler is invoked.

6.5 Evaluation

Two methods of studying address translation in NIs are simulation and hardware measurements. Hardware measurements are very accurate for the system under study, but are difficult to extrapolate to other systems and can be limited by constraints of the particular environment. Since we want to study a basic mechanism across design points, simulations are better suited to this task, being more flexible. However, I also present results from an implementation on real hardware to demonstrate the feasibility of the approach.

The goal in this evaluation is to demonstrate that for the designs discussed in Section 6.3, the desired performance properties hold, that is they are able to take advantage of locality and they will degrade gracefully when their capacity is exceeded. I do not attempt to determine whether NI address translation mechanisms can capture the locality present in typical workloads nor do I attempt to determine by how much the performance of those workloads can be improved with each design. Such study, interesting by itself, is highly dependent on the application domain and its programming conventions and beyond the scope of this thesis.

6.5.1 Simulation Framework

The simulation results were obtained using WWT-II, which is a detailed execution-driven discrete-event simulator [RHL+93,MRF+97]. The simulator is able to execute actual Sparc binaries by rewriting them to insert code that keeps track of the execution cycles. Each node contains a 300 MHz dual-issue SPARC processor modeled after the ROSS HyperSPARC and an NI similar to the CM-5 NI [Thi91], augmented to support coherent DMA and address translation. As on the CM-5, the NI resides on the memory bus. The original CM-5 NI supported packets up to twenty bytes long. This restriction has been lifted in the simulator where the simulated NI is capable of sending packets up to 4906 byte long. The characteristics of the network were chosen so that any performance degradation is strictly due to data transfers within the node (e.g., the network latency is only 100 CPU cycles). The detailed node parameters are listed in Table 6.2.

User-level messaging libraries model the messaging subsystem in Blizzard/CM-5 [SFL+94] (without the fragmentation support required due to the small packet size of the original CM-5 NI). The CPU constructs messages by writing with uncached stores to the output queue of the NI. It receives the messages with uncached loads from the input queue. For DMA operations, the packet format has been extended to include DMA transfer descriptors. The NI examines the headers of incoming or outgoing messages. When the appropriate field is set on the header, it extracts the local or remote virtual address and the data length from the transfer descriptors and performs the DMA transfer.

6.5.2 Simulation Results

Best Case Throughput & Latency. First, we will determine the effectiveness of minimal messaging in the best case. Therefore, we will measure the maximum possible benefit across

Table 6.2: Simulation Node Parameters

CPU Characteristics	300 MHz dual issue HyperSparc 1 MB direct mapped processor cache with 32 byte transfer block size
Memory Bus Characteristics	MBus level-2 coherence protocol[Inc91], 100 MHz, 64 bit wide
Memory Bus Bandwidth	800 MB/sec (peak) 320 MB/sec (32 byte block transfers) 200 MB/sec (uncached stores to the NI) 120 MB/sec (uncached loads from the NI)
Network Characteristics	100 CPU cycle latency per message Infinite bandwidth (limited by eight-message sliding window flow control protocol)

message sizes. We are interested in two metrics, throughput and latency. For the latency numbers, we pingpong a message between two nodes and measure the round trip time. For the throughput numbers, we blast messages from one node to another. Each message sends the same data buffer from the sender to the receiver. Therefore, we never miss on the translation structures after the first message. Moreover, I assume zero lookup overhead because I want to evaluate an upper bound on the potential of minimal messaging. .

The results (Figure 6-5) reveal the following three things about minimal messaging itself:

- For small data blocks, minimal messaging offers little, if any advantage. The gain for avoiding the extra copy is small and it is balanced by the extra overhead of writing the transfer descriptors and performing processor-coherent DMA operations (if the message data are not aligned to 32 bytes processor-coherent DMA requires read-modify-write bus transactions). It should be noted that this result is limited to user-messaging models in which like active messages, a handler is invoked for every incoming message. In simpler user-messaging models as for example those which support remote memory accesses, minimal messaging can enhance performance simply because the CPU is not in the critical path on the receiver and therefore, communication and computation can be overlapped.
- The limiting factor is the memory bus occupancy and not the bandwidth of DMA operations vs. uncached accesses (i.e., we have not tilted the experimental setup to favor block memory bus transfers). Indeed, using an intermediate buffer results in worse performance compared to directly accessing the NI with uncached memory operations since the data move through the memory bus three times.

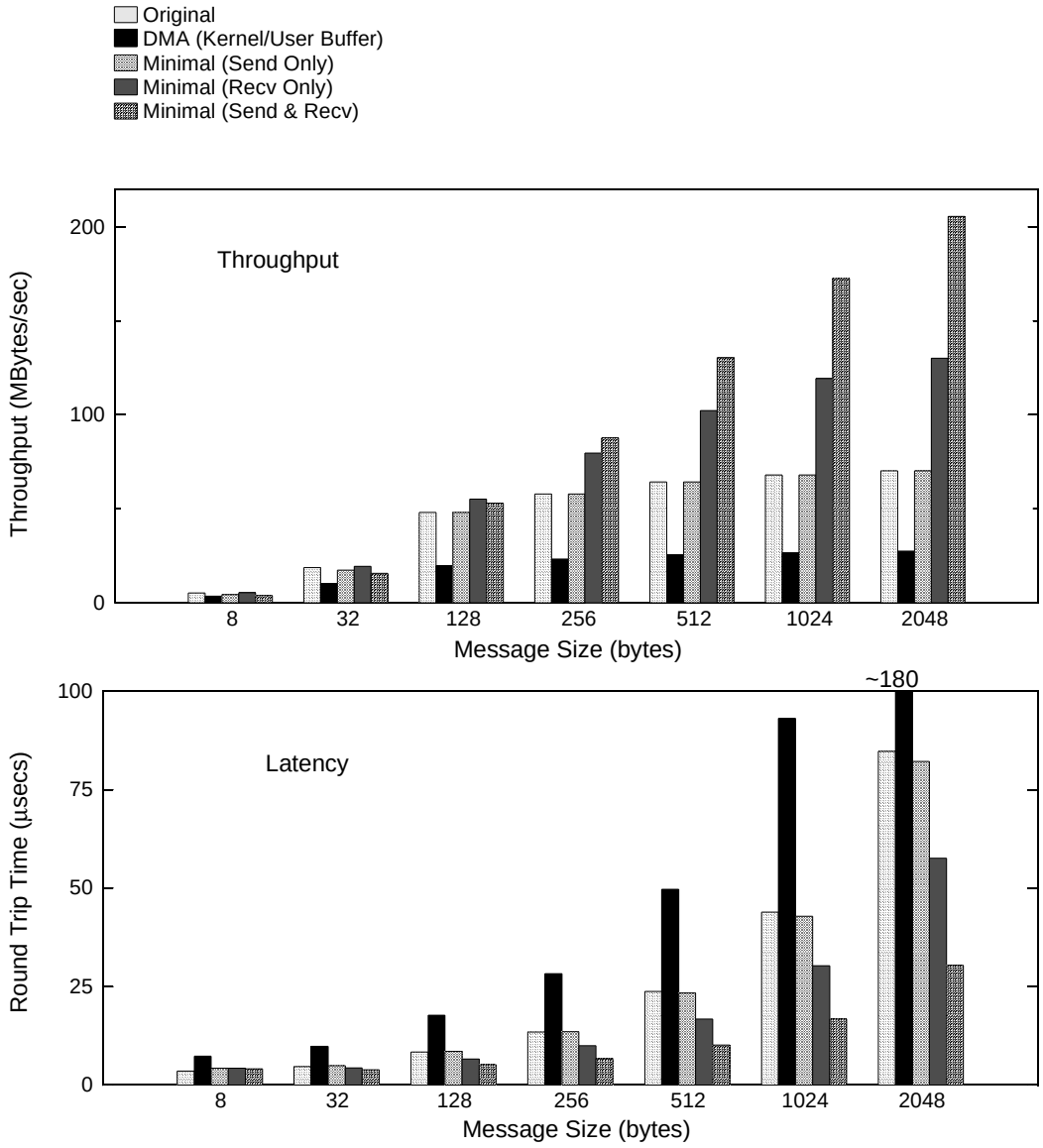

Figure 6-5. Simulated best-case throughput and latency.

Original *is the single copy method using uncached accesses to the NI.* ***DMA*** *is the single copy method using an intermediate buffer which the NI can access with DMA operations and the CPU with cacheable accesses. For the next three cases, I use* ***minimal*** *messaging on the send side only, on the receive side only or on both sends and receives, respectively. The lookup cost is zero cycles in all cases (including accesses to the intermediate*

- Minimal messaging offers enhanced performance because of the overhead in moving data through the CPU. When minimal messaging is used on both the sender and the receiver, it

buys more than what we gain by adding the benefits of doing it only on one side. The reason is that processor-coherent DMA transfers invalidate the data in the processor cache. When the CPU accesses them again for the next message, it incurs cache misses.

Lookup & Miss Behavior. The next step is to evaluate the performance of the designs discussed in Section 6.3, as the stress on the translation structures changes. We measure the throughput and latency as before but transferring *n* buffers each in a different page, for increasing *n's*. For small *n's*, (i.e., number of buffers), the performance is going to be determined by the lookup method. Therefore, we are measuring the best case scenario. For large *n's*, the translation structures will be stretched and start thrashing as they encompass a larger working set. Therefore, we are measuring the worst case scenario.

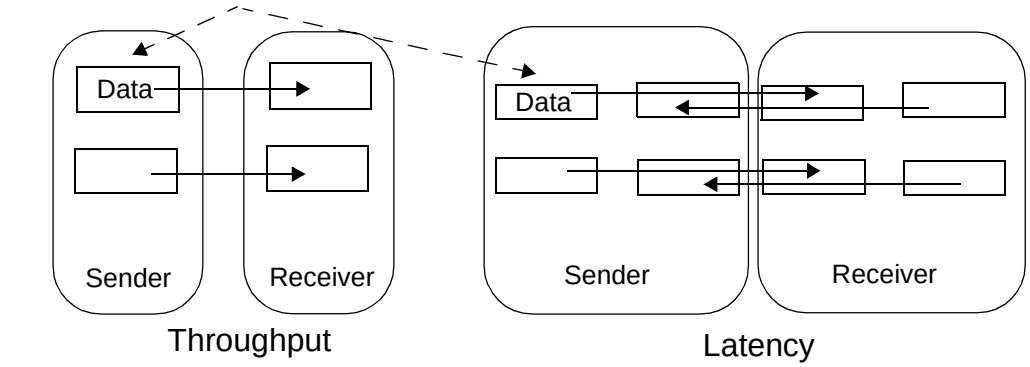

Figure 6-6. Message operations in the microbenchmarks.

Successive message operations iterate through N successive buffers spaced 4Kb apart. Therefore, every message originates and arrives in different memory pages. For N equal to one, these microbenchmarks measure the maximum possible benefit of minimal messaging (always hit). For N greater than the entries in the address translation structures, they will measure worst-case performance (always miss).

Figure 6-7 present the throughput results and Figure 6-8 presents the latency results from these experiments for two buffer sizes: 512 and 2048 bytes. *Original* uses uncached accesses to the NI. The rest have address translation structures with 32 entries. In all other cases, the size of the translation structures is 32 entries (this is too small for software structures, but it makes it easier to present the results). The replacement policy is random in all address translation designs with the exception of host lookup where it is an LRU policy. The detailed address translation parameters for each design are listed in Table 6.3.

For the lookup, I evaluate three alternatives: *(a) NI Hard Lookup* is a fully associative hardware structure on the NI with zero lookup cost, *(b) NI Soft Lookup* is a two-way associative

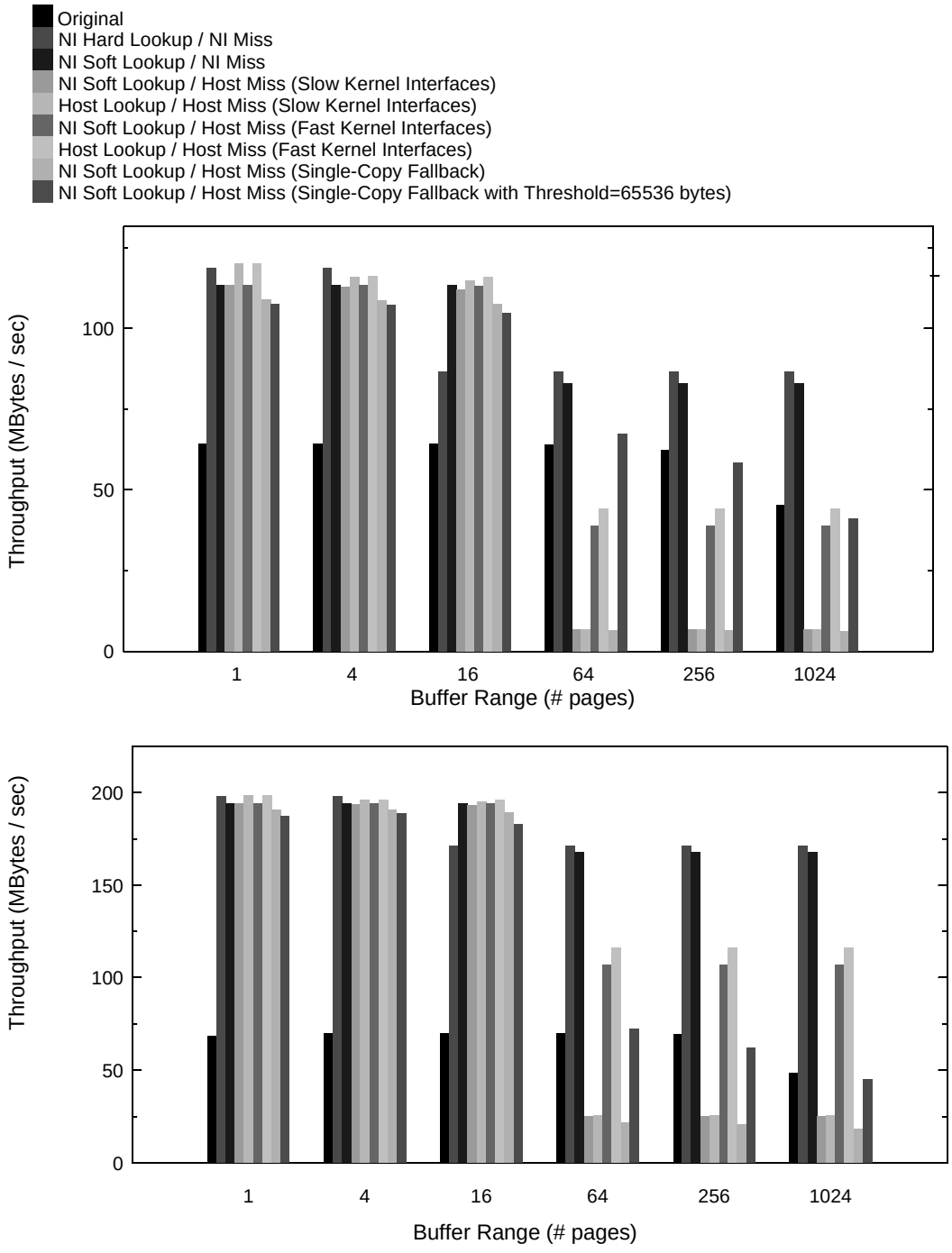

Figure 6-7. Simulated throughput as a function of the buffer range.

Table 6.3: Simulation Address Translation Parameters

Translation Cache Size	32 entries
Replacement Policy	Random (expect CPU Lookup -- CPU Miss Service where it is LRU)
Associativity	NI Hardware Lookup: Full NI Software Lookup: 2-way CPU Lookup: Full (three level page table)
Lookup Cost	NI Hardware Lookup: 0 cycles NI Software Lookup: 45 cycles per cache set CPU Lookup: ~180-500 cycles
Miss Service Cost	NI Miss: 450 cycles (3 * CPU TLB miss cost) CPU Miss (Slow kernel interfaces): 20000 cycles (+ 200 cycles trap cost) CPU Miss (Fast kernel interfaces): 2000 cycles (+ 200 cycles trap cost)

software structure on the NI with 45 cycles per set lookup cost, *(c) Host Lookup* is a user-maintained three-level page table structure on the host.

For the miss, I evaluate four alternatives: *(a) NI Miss* handles misses on the NI with a cost of 450 cycles (3 times the cost of a CPU TLB miss), *(b) Host Miss (Slow Kernel Interfaces)* handles the miss on the host CPU, triggered from the NI by an interrupt or the application through a system call and it is serviced by the device driver using standard kernel interfaces (200 cycles interrupt cost + 20000 cycles for the operation), *(c) Host Miss (Fast Kernel Interfaces)* handles the miss on the host CPU, triggered as in (b) but serviced within the kernel trap handler (200 cycles interrupt cost & 2000 cycles for the operation), *(d) Host Miss (Single-Copy Fallback)* uses uncached accesses when the lookup fails and deals with the miss after the message operation, and *(e) Host Miss (Single-Copy Fallback with Theshold=65366)* is like (d) but only handles one miss for every 64Kb of data transferred.

The results from these experiments demonstrate that while all the designs can take advantage of locality, only three classes of designs gracefully degrade. The designs that serve misses on the NI offer enhanced performance even with a miss for every message operation. The designs that use optimized kernel interfaces also show the same result but start from larger messages since the miss cost has increased. Finally, the designs that use the single-copy fallback with a threshold to control miss processing, degrade to within 10% of the single copy approach.

We determine the impact of the lookup method when we always hit in the translation structures. As expected, the graphs show that the lookup method does not affect the messaging performance significantly. Doing the lookup with hardware support is slightly better than using software but the difference is very small. However, software tables can easily contain thousands of entries compared to a few tenths included in a TLB. This indicates that there is no incentive for hardware lookup support (e.g., a network coprocessor with a powerful MMU). The host lookup method also performs well in this experiment. However, its performance slightly degrades as the number of translation entries increases. For realistic table sizes, the host CPU would experience cache misses in accesses to the translation structures.

We determine the impact of miss handling by making the translation structures thrash. The simulation results indicate:

- For both message sizes, handling the misses on the NI is fast enough so that minimal messaging remains more efficient than single-copy.
- Using fast kernel interfaces also reduces the miss cost. However, the overall performance when we exceed the limits of the translation structure with 512 byte messages is worse than the single copy approach.
- The single-copy fallback with a suitable threshold (one miss serviced for every 64KB data through the NI) tracks the performance of the single copy approach and therefore, degrades gracefully. Without these modifications in the miss policy, the performance plummets. Handling the misses on the CPU with slow kernel interfaces results in thrashing behavior and the single-copy fallback fails to offer any benefits by itself. In fact, it makes things worse since we have to service the miss and use the slower single-copy path.
- The performance of the single-copy approach is also sensitive to the number of buffers. It drops as the size of the buffer exceeds the capacity of the CPU cache which again demonstrates the cost of moving data through the CPU.

6.5.3 Blizzard Framework

To make the feasibility case for the approach, I will present results from the Blizzard implementation that actually predated the simulation work. In COW, the NI is located on the I/O bus (SBUS [Mic91]) which supports processor coherent I/O transfers between the memory and I/O devices. The bus bridge supports an I/O memory management unit (IOMMU) which provides translation from SBUS addresses to physical addresses. For user level messaging, the device driver maps the NI's local memory, which is accessed with uncached memory operations, in the user address space.

As we saw in Chapter 3, the NI is controlled by a 5 MIPS 16-bit processor, which runs a control program that implements separate send and receive queues on the NI memory. Each queue has 256 entries with headers that point to the message data. Under the standard I/O architecture, the NI can use DMA only for kernel addresses. Therefore, to permit DMA to the user address space, the driver maps a kernel I/O buffer in the user address space, which is used for the message data. For minimal messaging, we can allow the queue entries to contain user virtual addresses for NI lookup or table indices for host lookup. The user code cannot directly

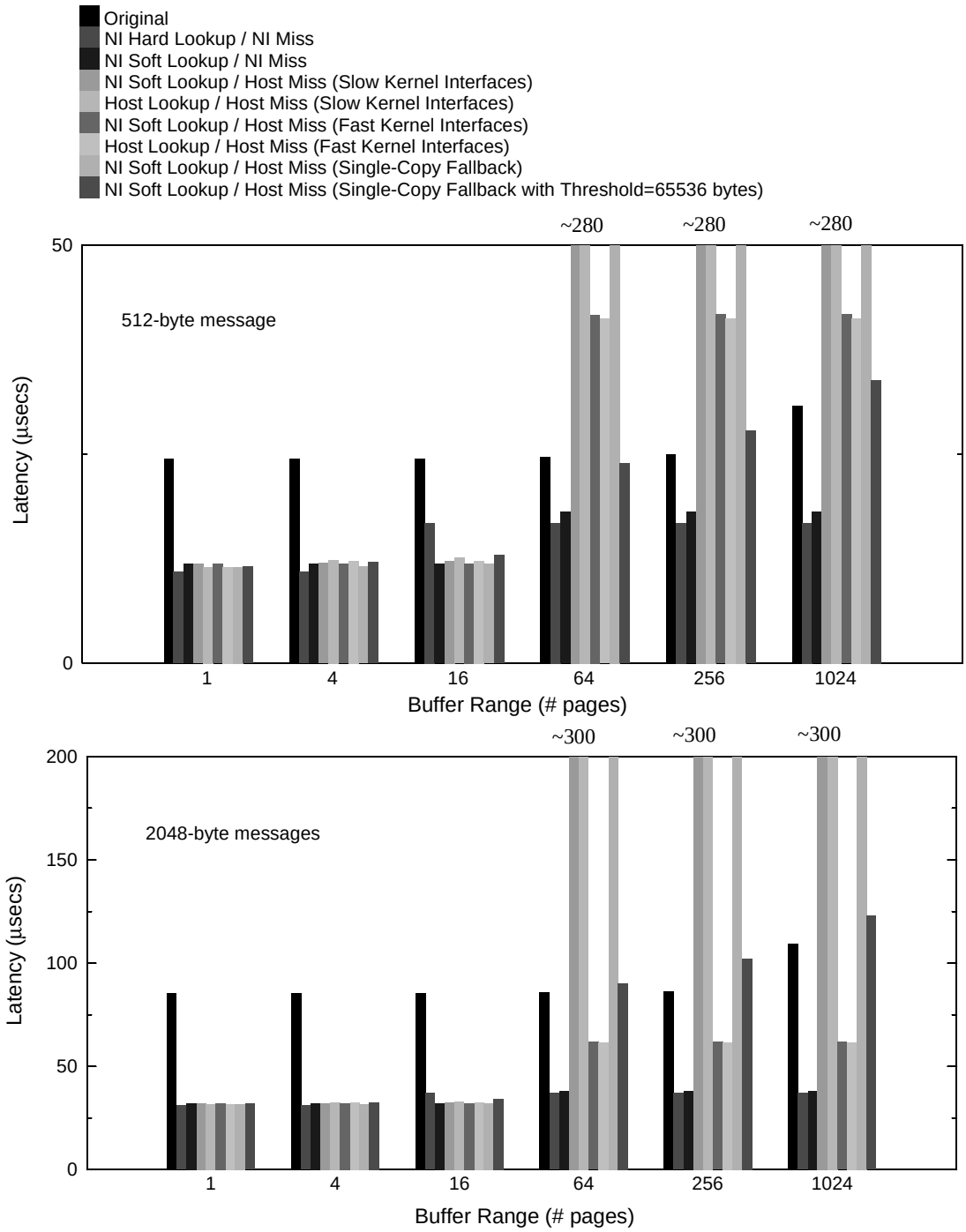

Figure 6-8. Simulated latency as a function of the buffer range.

specify physical addresses. Instead, the device driver has been modified to install physical addresses in protected, kernel accessible NI memory. Less than two hundred lines of C code in

the NI control program and device driver were added to support address translation. To perform the actual transfer, the BYPASS mode of the SBUS-to-MBUS bridge is used to directly access the physical memory of the application address space. With this mode, intelligent peripherals can use physical addresses *bypassing* the IOMMU translations. The detailed node parameters are listed in Table 6.4.

Table 6.4: Myrinet Node Parameters

CPU Characteristics	66 MHz dual issue HyperSparc 256 MB direct mapped processor cache with 32 byte transfer block size
Memory Bus Characteristics	MBus level-2 coherence protocol[Inc91], 50 MHz, 64 bit wide
I/O Bus Characteristics	25 MHz 32 bit SBus
Memory Bus Bandwidth	400 MB/sec (peak) 160 MB/sec (32 byte block transfers) 20 MB/sec (through I/O bus bridge)
Network Characteristics	17 μsecs latency (for messages without data) 40 MB/sec bandwidth (limited by eight-message sliding window flow control protocol)

6.5.4 Blizzard Results

In Blizzard, on the sender the lookup is performed on the host. On the receiver, the lookup is performed on the NI. In both cases, the single-copy fallback is used. Due to the Myrinet design, this is a requirement on the receive side; if a receiver stops accepting messages for a small amount of time, the sender causes the receiver board to reset itself. The translation structures contain 256 entries each. For both NI and host lookup, the lookup cost is ~2 μsecs while the miss cost is ~100 μsecs using standard kernel interfaces that wire pages in memory.

Figure 6-9 presents the best-case results across message sizes and Figure 6-10 presents the stress results. The results show that: (i) in the best case, minimal messaging reduces latency by up to 25% and 40% for 512-byte and 2048-byte messages, respectively. (ii) in the worst case, it increases latency by 9% and 2% for 512-byte and 2048-byte messages, respectively, when we control the rate of miss processing. Overall, the latency results show similar trends as in the simulator.

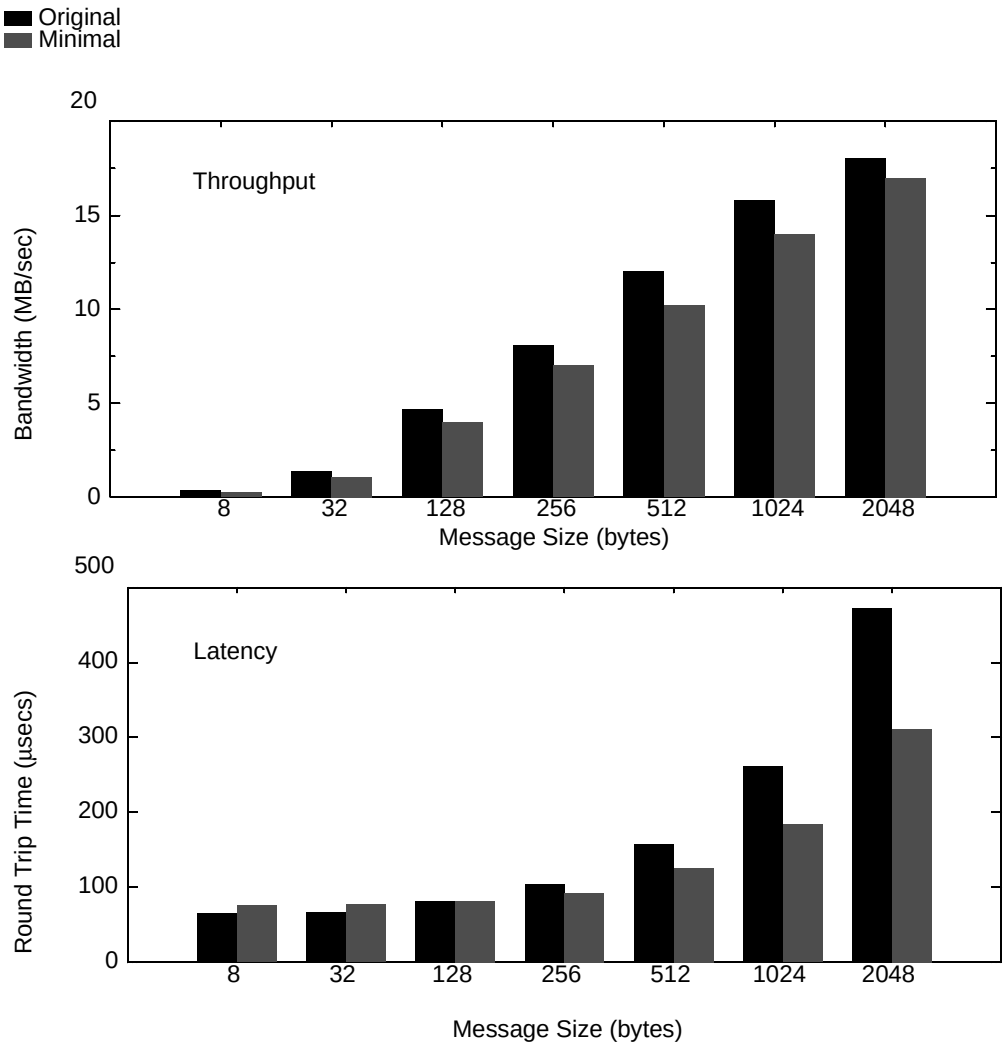

Figure 6-9. Myrinet best-case throughput and latency.

Original uses the shared user/kernel buffer for the message data. Minimal accesses directly the application data structures. On the send side, the lookup is performed on the host. On the receive side, the lookup is performed on the NI.

Unlike the simulator, throughput does not increase. In these machines, there is limited bandwidth across the SBus/MBus bridge. In the single copy approach copying the data to the intermediate buffer is overlapped with the transfer across the bridge, which is the bottleneck. With minimal messaging, the CPU is free to poll for incoming messages across the bridge which slightly reduces throughput by 15% and 2% for 512-byte and 2048-byte messages.

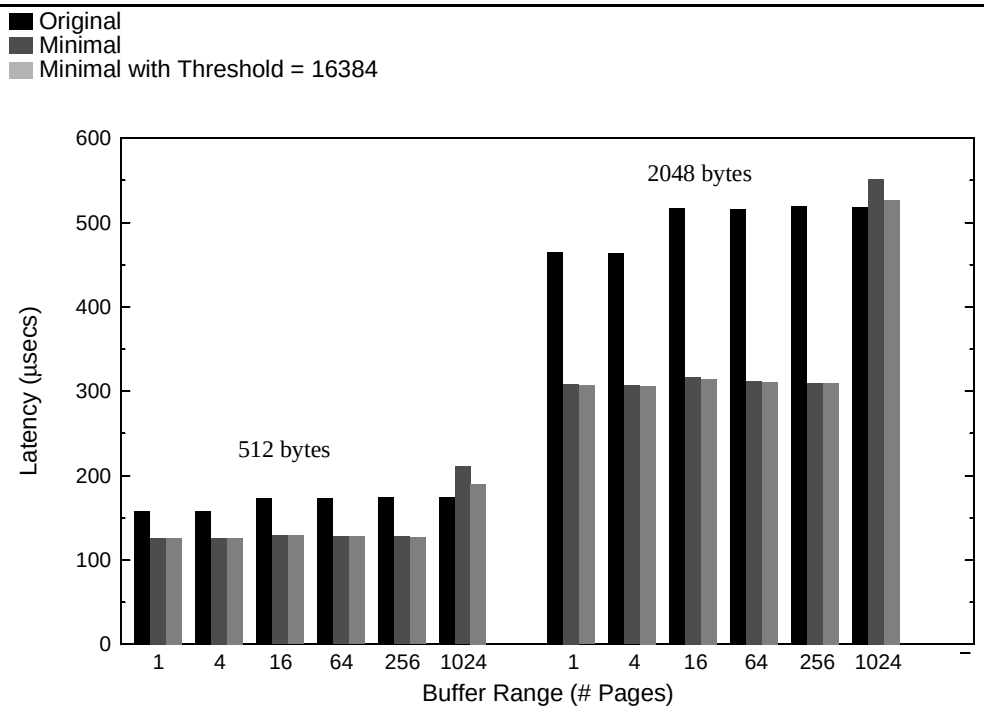

Figure 6-10. Myrinet latency as a function of the buffer range.

Original uses the shared user/kernel buffer for the message data. Minimal accesses directly the application data structures. On the send side, the lookup is performed on the host. On the receive side, the lookup is performed on the NI. The translation structures on the send and the receive side support 256 entries each. In both cases, the single-copy fallback method is used. "Minimal with Threshold = 16384" services only one miss for every 16384 bytes transferred.

6.6 Related Work

Abstractions that provide *sender-based communication* such as *HP Hamlyn* and *Berkeley Active Messages***,** are powerful enough to support minimal messaging. The sender specifies the source and destination addresses as offsets in message segments for every message. In theory, you can define message segments that cover the entire application address space. However, in current Hamlyn [BJM+96,Wil92] and Active Messages [Mar94,LC95,CLMY96] implementations the address translation structures are not there or are limited to their reach. For example, the latest prototype Hamlyn implementation [BJM+96] is built on hardware identical to ours (Myrinet), yet message buffers must be pinned in main memory. This limits the coverage of the mechanism to the amount of application data that can be wired in physical

memory. Similarly, the Active Messages implementation on Myrinet uses a single-copy approach through an intermediate shared user/kernel buffer where the NI pulls or pushes message data.

Arizona ADCs [DPD94] have been designed to optimize stream traffic. In Section 6.2, we discussed why this design cannot fully support minimal messaging. The base *Cornell UNet* [vEBBV95] architecture supports an abstraction similar to ADCs, and therefore has the same limitations as ADCs. In the original UNet paper [vEBBV95], a direct access UNet architecture is discussed that includes communication segments able to encompass all the user address space but the architecture is restricted to future NI designs. Recent work [BWvE97] attempted to incorporate address translation mechanisms for existing NIs. Nevertheless, the ADC abstraction has not changed and therefore, the designs are unable to move data to their final destination without extra copying.

Mitsubishi DART [OZH+96] is a commercially available NI that comes close to properly support minimal messaging. DART core has been designed to support ADCs including sophisticated address translation support. Moreover, it defines an interface to a separate coprocessor that process messages. Presumably, it will be used to support the hybrid deposit model [Osb94], an abstraction similar to Active Messages, in which the data destination is a function of the receiver's and t sender's state. Unfortunately, the address translation is geared towards ADCs. As a result, the translation structures are not flexible enough to efficiently support minimal messaging. The host CPU is always interrupted to handle misses while the message is blocked until the miss can be resolved. Furthermore, there are not any provisions for a fallback action. Thus, the design requires fast kernel interfaces to gracefully degrade once the translation limits are exceeded.

Designs with a *network coprocessor*, like *Meiko CS* [BCM94] and *Intel Paragon* [Int93] can support minimal messaging using the microprocessor's address translation hardware and a separate DMA engine. Nonetheless, address translation mechanisms implemented for CPUs (TLBs) are not always appropriate for NIs. There are two potential problems. First, the reach of a CPU TLB is very small, typically a few dozen of pages. Message operations can span over a wide range of addresses, which is much larger than what TLBs can cover. Moreover, the data transfers compete with other memory accesses (kernel instructions, kernel data, I/O addresses), effectively making the TLB miss the common case for any message operation. Second, data transfers from/to the user address space require the CPU to switch the hardware context to the appropriate process. This operation can have significant overhead depending on the coprocessor's architecture (e.g., number of CPU hardware contexts, TLB and/or the cache flushing). Alternatively, the coprocessor can access page tables in software making the coprocessor TLB useless for minimal messaging.

In these designs that support *remote memory accesses*, memory pages in the sender's address space are associated with memory pages in the receiver's address space. Memory accesses on the sender are captured by the NI and forwarded to the associated page on the receiver. Page associations are either direct (the sender knows the remote physical address) or

indirect (through global network addresses). Examples of this approach include *Princeton SHRIMP* [BDFL96], *Forth Telegraphos II* [MK96], *DEC Memory Channel* [GCP96] and *Tandem TNet* [Hor95]. SHRIMP and Telegraphos II use direct page associations. The Memory Channel and TNet use indirect page associations. Common characteristic of these designs is their inability to handle misses in the translation structures. Therefore, the translations must in place before messaging operations, which requires communication pages to be locked in memory. Moreover, changing the reach of the translation mechanisms requires expensive system calls. SHRIMP's prototype NI can hold up to 32K of associations between pages. Thus, minimal messaging is supported for up to 128Mb of application data from every sender. Similarly, the Memory Channel supports up to ~50K pages. In TNet, remote memory operations are supported in a 32-bit window to a node's physical memory.

A class of designs supports minimal messaging by using the CPU in kernel mode to instruct the NI to move the data to the appropriate place. Such approaches include page remapping in the kernel (implemented in Solaris 2.6 TCP [kJC96]), Washington's in-kernel emulation of the remote memory access model [TLL94] and other VM manipulations [BS96]. In these systems, minimum messaging is achieved if the NI can directly access the main memory. However, the kernel is involved in every transfer and thus, user-level messaging is not supported.

Princeton User-level DMA (UDMA) [BDFL96] avoids OS intervention and supports minimal messaging when it is used both to send and receive messages. UDMA is used in SHRIMP to initiate DMA transfers. In this case, it supports minimal messaging on the sender but on the receiver, it suffers from the same problems that we have discussed for SHRIMP. Nevertheless, it can be used on both the sender and the receiver (without SHRIMP's support for remote memory accesses). Consequently, it supports minimal messaging in a way that shares common features with the design that allows user-controlled mappings (Section 6.3.3). In the common case, both avoid kernel intervention for data transfers and in both the CPU is in the critical path for message operations. Unlike our design, UDMA requires hardware support in NIs to capture transfer requests.

Cray T3E [Sco96] combines remote memory accesses with an approach similar to UDMA. It supports minimal messaging through special NI registers. The CPU initiates transfers directly from remote memory to the NI registers on the local node. It subsequently initiates the transfer of the data from the NI registers to local memory. The CPU must be involved once for every 64 bytes transferred (the maximum message size supported). T3E includes extensive hardware support for address translation in the form of complete page tables that describe global communication segments. However, the page tables must always have valid translations, and therefore the communication pages are wired in memory.

6.7 Conclusions

I have argued that minimal messaging requires address translation support in the NI. The address translation mechanism should have a consistent interface, be able to cover all the user address space, take advantage of locality and degrade gracefully.

I have classified the address translations mechanisms according to where the lookup and the miss handling are performed. This classification defined a design space, which we systematically analyzed. Furthermore, it exposed a design point for which I proposed a novel interface for user-controlled mappings. We examined the OS requirements for each design point and discussed techniques to guarantee that the designs gracefully degrade in the absence of appropriate OS interfaces.

The simulation results validated that even without appropriate OS interfaces we can take advantage of locality and degrade gracefully. Moreover, the same results indicated that for the performance of the address translation structures more attention must be paid on how misses are handled than how the lookup is performed. To summarize:

- Hardware lookup structures in the NI are not required since simple software schemes are fast enough.
- We would prefer the NI to handle misses, for performance and clean integration with the OS.
- If this is not possible due to lack of OS support, minimal messaging should be considered an optimization with the default case being the single-copy method. It will help when possible but it should not thrash when the hardware limits are exceeded.
- Fast kernel interfaces are a short-term solution; even if we disregard the poor OS integration, as CPUs get more complicated, the trap cost is going to rise.
- Finally, experimental results from real hardware demonstrate the feasibility and potential of this approach.

As a final conclusion, the practical meaning of this study for hardware designers is that the key principle in the design of address translation mechanisms should be flexibility to support all possible levels of OS integration during the lifetime of the design.

Chapter 7

Conclusions

Finally, faithful (or impatient) reader, we have reached at the end of this thesis. In this last chapter, Section 7.1 presents a summary of the thesis, Section 7.2 discusses the implications of this study and Section 7.3 speculates on future directions for this work.

7.1 Thesis Summary

This thesis explored fine-grain distributed shared memory (FGDSM) systems on clusters of workstations to support parallel programs. FGDSM systems rely on fine-grain access control to selectively restrict reads and writes to cache-block-sized memory regions. The thesis explored the issues involved in the design and implementation of FGDSM systems on clusters of commodity workstations running commodity operating systems. It presented Blizzard, a family of FGDSM systems on the Wisconsin Cluster of Workstations (COW). Blizzard supports the Tempest interface that implements shared memory coherence as user-level libraries.

Chapter 2 investigated fine-grain access control implementations on commodity workstations. It presented four different fine-grain tag implementations with different performance characteristics. Blizzard-S adds a fast lookup before each shared-memory reference by modifying the program's executable. Blizzard-E uses the memory's error-correcting code (ECC) bits as valid bits and the page protection to emulate read-only bits. Blizzard-ES combines the two techniques using software lookups for stores and ECC for loads. Blizzard/T uses Vortex, a custom fine-grain access control accelerator board, and it exploits Vortex's ability to support hardware fine-grain tags modified without kernel intervention. The chapter identified fine-grain access control as a memory property that can be associated either with the physical or the virtual memory. It argued that fine-grain access control as a property name should be asso-

ciated with the virtual address space for flexibility while it should be associated with the physical address space as a property content for performance. It described the design of the kernel support required for hardware techniques within a commodity operating system. Blizzard's kernel support offers extended kernel functionality through runtime-loadable modules, follows a modular approach, and optimizes all the performance-critical operations. It presented low-level performance results to evaluate the different Blizzard fine-grain access control implementations.

Chapter 3 investigated messaging subsystem design for FGDSM systems. For standard shared memory coherence protocols, low-latency communication is critical to the performance of many parallel programs. For application-specific protocols, optimized to fit the application communication requirements, high bandwidth is of equal concern to low latency. Tempest's messaging interface is based on the Berkeley active messages but it differs in two important aspects, necessary to use Tempest messages to implement *transparent* shared memory protocols. Tempest messages are not constrained to follow a request-reply protocol and are delivered without explicit polling by an application program. The chapter presented a methodology to compute the buffer requirements for an arbitrary messaging protocol. Using this methodology, it showed that shared memory coherence protocols, while not pure request/reply protocols, have bounded buffer requirements. It evaluated buffer allocation policies for active message implementations. It proposed a buffer allocation policy that does not require buffers in a node's local memory in the common case, yet is robust enough to handle arbitrary traffic streams. It evaluated different solutions to avoid explicit polling and it proposed implicit polling using binary rewriting to insert polls in the application code. It presented performance results that indicate that the extended functionality does not affect performance adversely.

Chapter 4 evaluated Blizzard as a platform for parallel applications. For this task, both low-level microbenchmarks and application benchmarks were used. The performance results showed that parallel applications with small communication requirements achieve significant speedups using transparent shared memory. Parallel applications with high communication requirements however, require application-specific coherence protocols to harness the available message bandwidth.

Chapter 5 investigated extending FGDSM systems to clusters of multiprocessor workstations. FGDSM systems are particularly attractive for implementing distributed shared memory on multiprocessor clusters because they transparently extend the node's fine-grain hardware shared memory abstraction across the cluster. Grouping processors into multiprocessor nodes allows them to communicate within the node using fast hardware shared-memory mechanisms. Multiple processors can also improve performance by overlapping application execution with protocol actions. However, simultaneous sharing of node's resources between the application and the protocol requires mechanisms for guaranteeing atomic accesses. Without efficient support for atomicity, accesses to frequently shared resources may incur high overheads and result in lower performance. The chapter identified the shared resources in FGDSM systems to which access should be controlled in an multipro-

cessor environment and it proposes techniques to address the synchronization issues for each resource based on the frequency and type of accesses to that resource. It compared the performance of Blizzard's uniprocessor-node and multiprocessor-node implementations while keeping the machines' aggregate number of processors and amount of memory constant. The performance results suggest that: (i) grouping processors results in competitive performance while substantially reducing hardware requirements, (ii) grouping benefits from custom hardware support for fine-grain sharing, (iii) grouping boosts performance in FGDSM systems with high-overhead protocol operations, (iv) grouping can hurt performance in pathological cases due to the ad hoc scheduling of protocol operations on a multiprocessor node.

Chapter 6 investigated address translation mechanisms in network interfaces to support zerocopy messaging directly to user data structures and avoid redundant data copying. Good messaging performance is important for application-specific protocols that push the limits of the messaging hardware. For zero-copy messaging, the network interface must examine the message contents, determine the data location and perform the transfer. The application accesses data using virtual addresses, which can be passed to the interface when the message operation is initiated. However, the network interface is a device that it accesses memory using physical addresses. Therefore, the virtual address known by the application must be translated to a physical address usable by the network interface; hence, an address translation mechanism is required. The chapter presented a classification of address translation mechanisms for NIs, based on where the lookup and the miss handling are performed. It analyzed the address translation design space and it evaluated a series of designs with increasingly higher operating system support requirements. For each design point, it determined whether the design can take advantage of locality and whether it gracefully degrades once the translation structures start threashing. Moreover, it considered the required operating support and proposed techniques to make these properties hold in the absence of appropriate operating system interfaces. It provided performance data from simulations that demonstrated that even without operating system support, there exist solutions so that the performance properties hold. It demonstrated the feasibility of the approach by presenting experimental results from an implementation within the Blizzard framework.

7.2 Implications

The focus of my work has been to integrate various ideas to a functional, complete FGDSM system. Moving from ideas to concrete systems requires a perceptual change of focus. Every aspect of a good design must be evaluated not by itself, but rather on how well it fits within the whole system. With regards to this goal, I demonstrated the feasibility of FGDSM systems on clusters of commodity workstations running commodity operating systems by presenting Blizzard, a family of FGDSM systems on Wisconsin COW.

The most important lesson for system designers is that FGDSM systems on cluster of workstations can efficiently support a large range of shared memory parallel applications. This result does not come at odds with hardware-centric approaches to shared memory. On the contrary, it complements these approaches because it demonstrates the portability of shared mem-

ory applications to messaging passing environments. This is especially important since the scalability of hardware shared memory is limited either by technology and cost constraints or high availability considerations. Future large-scale parallel platforms are more likely than not to resemble a cluster of hardware shared memory machines.

Often, the largest obstacle in porting shared memory applications in message passing environments is the extensive redesign of algorithms and data structures that this effort requires. Software FGDSM systems address the portability issue, as do page-based DSM systems, which however, require weaker memory consistency models for acceptable performance with fine-grain applications. Moreover, Blizzard supports the Tempest interface that allows selective use of application-specific coherence protocols to exploit the raw messaging performance. This is particularly important for software FGDSM systems due to the relatively large fine-grain access control overheads.

This study explored system issues involved in the design and implementation of FGDSM systems. First, it showed that there is a abundance of mechanisms in commodity workstations that can be used to implement fine-grain access control. Software methods offer good portability and relatively good performance, an argument collaborated by Digital's Shasta. Hardware methods can offer better performance if the fine-grain tags can be manipulated from user level. Moreover, this study demonstrated that kernel support for hardware-based approaches can be incorporated within commodity operating systems. Second, this study addressed the issues related to the design and implementations of high performance messaging subsystems for FGDSM systems. It demonstrated that it is possible to design efficient subsystems to support the generalized semantics dictated by shared memory coherence protocol. Third, this study demonstrated that multiprocessor workstations can be effectively used as building blocks in FGDSM systems offering competitive performance with uniprocessor-node implementations. Fourth, this study argued that messaging performance can be optimized by using address translation mechanisms in network interfaces to support zero-copy protocols. Furthermore, it discussed how such mechanisms should be implemented and integrated within the operating system.

7.3 Future Directions

Blizzard represents a research prototype. It served as a vehicle to demonstrate the feasibility of FGDSM systems on clusters of workstations, evaluate their potential as parallel platforms and in general, explore research ideas. However, much work remains both to complete the exploration of the design space and to design production-quality FGDSM systems. Two particular areas that appear promising for further investigation are fault tolerance issues in FGDSM systems and multiprogramming support in clusters of workstations (virtualization).

7.3.1 Fault Tolerance

For many applications, the ability to survive system failures is important. In this section, I discuss two examples, with regards to transient message errors and process/node faults, of

appropriate fault tolerance support for parallel applications. In general, designing applications that survive any system failures is not a trivial problem. Indeed, a fault can appear in any component of a system. Typically, the system support ranges from little to full fault tolerance support, with most systems being at the extremes of this range. However, in the context of FGDSM systems, the focus should be on providing the required system support so that each application can choose its fault tolerance properties, depending on the extra overheads it can tolerate. For example, for some applications it might suffice to be able to detect transient message errors while others require more strict fault tolerance guarantees. It cannot be overemphasized that only applications which require enhanced fault tolerance support should have to pay additional overheads.

The objective for such work should be identify the types of system failures, suggest ways to deal with them in the context of a fault tolerant application, and provide the necessary mechanisms to support such applications. While little research exists on fault tolerant issues in hardware shared memory [CRD+95,MGBK96] fault tolerance has been an important research area in distributed systems. The components of such systems use messaging primitives to communicate. Software DSM systems are particularly suited to take advantage of this research because software shared memory is also implemented on top of messaging. Furthermore, Blizzard's flexibility in supporting application-specific protocols as user-level libraries, facilitates the design of protocols with fault tolerant characteristics.

Transient Message Errors. Messages can get lost or corrupted as they travel though the network. Depending on the exact fault rate, we have two options; hide these faults inside the network protocols or expose them to the Tempest level. If they are common enough, so that no protocol can be designed without having to deal with them, we should take care of the problem in the network layer. On the other hand, if it is very unlikely for most applications to encounter them, only the applications with high reliability requirements should need to deal with them. This is likely to be the case when system areas networks [Bel96] are used to connect the cluster nodes. In this case, we can define an interface through which transient message faults are reported to the application. In other words, the responsibility of the system is to provide fault detection to the applications that wish to deal with these faults. In turn, applications with high reliability requirements can use cautious shared memory protocol that keep a copy of every message until safe remote receipt of the data is confirmed. For example, SCI [GJ91] is a shared memory coherence protocol that has been designed to deal in this manner with transient transmission faults.

Process & Node Faults. The cluster environment comprises of many workstations, able to operate independently. As long as the network connectivity is not severed, the workstations can continue to operate unhindered from events like node failures. Of course, this is not true for any applications that might have processes running on those nodes. Another class of faults includes process faults, which occur because of a bug in the user code, the parallel system, the operating system or even due to temporary resource shortage (for example, a request for memory from the system which cannot be satisfied at that time). Node faults can be either transient of permanent. Transient node faults mean that the node eventually is going to be rebooted and

the processes can be restarted. In that sense, they do not differ from process faults. Permanent node faults may result in the inability to access any stable storage local to the node that failed unless provisions are taken for redundant access paths to that storage. For example, in many current commercial offering for high availability, disks are shared among the cluster nodes.

From the perspective of the application there is little difference. whether it was a node or a process fault. The effect in both cases is the same: part of the computation state is lost. Depending on the application requirements, it might be adequate to guard for the availability of the shared address space, with or without data integrity. Alternatively, it might be important to be able to completely recover to a consistent state following process or node faults. In order to guard against such events, these applications need to periodically checkpoint their state, in a consistent manner. Then they can restart from the earliest checkpoint and recover from the fault. Various approaches have been proposed to offer checkpointing and recovery services in distributed systems, in general, and distributed shared virtual memory systems, in particular [KT87,WF90,TH90,MP97]. This work needs to be evaluated and extended in the context of the fine grained distributed shared memory systems.

7.3.2 COW Virtualization

In designing a system able to execute parallel programs, one must take into account what the user has come to expect in any computer system. While in a cluster of workstations each node is running a complete OS, which can locally provide the kind of a support that a user expects, little is available to integrate the services to the cluster level. The situation is likely to improve as commodity operating systems timidly start introducing clustering features (Microsoft Wolfpack [Lab97], Sun Solaris MC [KBM+96]). Many issues need to be addressed, however, before the integration becomes seamless at the cluster level. In this section, I discuss issues relevant to FGDSM systems.

Relaxing the single user assumption, represents a conceptual change of focus in the examination of the system. The most important metric ceases to be the speed of the parallel application. Instead, the system throughput starts to dominate in the discussion. For example, solutions that can offer more throughput even if they do not provide complete isolation to the parallel applications might be acceptable in this context. The first part of virtualizing an environment is containment among the parallel applications; no malicious application should compromise the others. The second part of the problem is equivalent to determining the allocation policies of the fixed system resources to the users, in a way that will treat the cluster as one machine. These resources include the processors, the network and the physical memory. The resource demands of the applications are clearly interdependent and no solution can be provided in a strictly local manner.

Scheduling. The scheduling policies affect every aspect of the resource allocation problem in a parallel environment since the choice of an allocation policy for this resource has direct consequences on the demands for other resources.

The two important dimensions along which we can assign processing resources to applications include space and time, i.e., space-sharing and time-sharing. The relationship between these policies is hierarchical in the sense that in any given system, the available nodes can be space-shared in subclusters. In turn, every subcluster can be allocated to one application, scheduled among parallel applications or time-shared in the traditional way with each node independent of the others. The exact choice would depend on the demands of the workload destined to run on any specific subcluster.

Space-sharing is particularly attractive for research prototypes because it allows to conduct experiments without interference from other activities while many users can work concurrently on a portion of the machine. For this reason, DJM supports space-sharing in Wisconsin COW. However, this policy is not appropriate for production environments where the users expect to make most of their hardware investment. Therefore, time-sharing is an issue worth investigating further.

An aggressive approach to support time-sharing, which has been implemented in the Thinking Machines CM-5, is to use gang scheduling (also called coscheduling). All the nodes and the network are concurrently changing context. In effect, for the duration of a quantum, the user has complete possession of the cluster resources. The importance of coscheduling has been noted very early in the history of such systems [OSS80]. The same result has been demonstrated in the context of shared memory parallel programs [BHMW94] executing scientific codes. Gang scheduling, however, is a very restrictive model. Moreover, it is very difficult to strictly implement it in a distributed environment without extensive modifications in the kernel philosophy. In addition, in such environments the cost of a parallel context switch can be prohibitive without hardware support.

It is possible to implement approximate coarse-grain gang scheduling for cooperating applications, using only the facilities already provided by modern operating systems. This can be achieved by a set of coordinating network daemons which can implement the gang scheduling policy starting and stopping the processes of a parallel application. Such an approach has appeared in Intel’s Paragon. It is obvious that the network daemons need to know the identity of the processes in a parallel application. The attractiveness of the scheme is that it requires no significant kernel modifications. However, it does require coordination with the system components that are crucial to the performance of a parallel application (e.g., network resources).

In general however, coscheduling is not appropriate for the mixture of interactive or batch, serial or parallel that are expected to be concurrently executing in the cluster. Recently, researchers have focused their attention and attempted to address exactly this scheduling problem. Emerging proposals include implicit scheduling [DAC96,ADC97] and dynamic coscheduling [SPWC97]. In implicit scheduling, processes relinguish control of the processor if they block waiting for a message more than some interval, statically or dynamically determined. In dynamic coscheduling, the system manipulates the priority of the processes to effect coscheduling based on observed traffic patterns.

Network Allocation. Any virtualization attempt must guarantee protection and forward progress. One application should not be able to impersonate another, in sending or receiving messages, and it should not be able to deadlock the network through misbehaving. Moreover, the system must provide fair allocation of the network resources, that is, allocation of the available bandwidth and the system buffers which will be used for the messages.

There is an evident interaction with the scheduling policies. In effect, a gang scheduling policy means that all the available bandwidth can be allocated to the currently running parallel application. However, this does not solve the problem of limited systems buffers. Typically, these buffers correspond to limited physical memory present at the host network interface card or limited kernel buffers.

The simplest policy for the allocation of the network resources is to provide fixed resources to every application. Effectively, the network is included among the resources to be gang scheduled. This can be achieved by defining a software interface between the parallel scheduler and the host network interface. Unfortunately, this scheme results in very poor utilization of the network; nobody but the application currently running can use the network. To achieve higher throughput, it may require to leave the network unscheduled, much like it is being used in traditional operating system. However, the exact policies must make sure that no undue burden is placed in the current application. For this purpose, it is required to take into account dynamic patterns and adapt to actual demand for the network resources.

The virtualization of the network must be done so that the limited physical resources do not require the kernel to be involved during the communication for enforcement. Effectively, it is desirable to take the kernel out of the critical path when resources are not scarce and use exception mechanisms to indicate resource shortages to the kernel. A requirement for such a scheme is to provide firewalls between applications so that misbehaving applications do not degrade the performance of the rest. In addition, the application interface defined for network access must allow user access to the network without kernel intervention. However, protection policies still need to be enforced. To do this efficiently, the required protection policies must be enforced using a hardware protection mechanism. In modern systems, the only available option is the virtual memory system. Thus each process wishing to access the network should contact the kernel for permission and the kernel will map in its address space the required interface memory and registers. Thereafter, the network interface must enforce the access rights when it sends or receives messages. Such models have been defined in detail within the context of Berkeley’s Active Message specification [MC95] and Intel’s Virtual Interface Architecture [DR97].

Memory Allocation. Shared memory coherence protocols such as Stache that implement a simple cache-only memory architecture (S-COMA) in software, pose an interesting dilemma on what to do when shared memory becomes overcommitted on a particular node. In particular, there are two alternatives. The first one is simply to do nothing and rely on the standard local kernel paging algorithms to move remote pages to secondary storage. This is possible in software FGDSM systems because the shared address space is build in user level, on top of

virtual memory. The second one is to flush the blocks in such pages in the home node and then explicitly deallocate the page. However, this alternative requires that the protocol itself monitors the page usage and implements some policy to determine candidate pages. Moreover, it requires a global policy to determine memory pressure, which may involve some interaction with the operating system running on the nodes. There are little evidence to suggest which of these two alternatives is preferable, whether local or global policies are more effective.

Appendix A

Blizzard Implementation

This appendix describes the Blizzard implementation on the Wisconsin Cluster of Workstations (COW). It is intended to be a self-contained document that presents the details of this implementation. While this information has been presented elsewhere in the main chapters of the thesis or in other relevant references, it is included here to allow someone to quickly get a first exposure to Blizzard.

The appendix is organized as follows. Section A.1 presents general information on the COW environment and in particular, it describes the hardware platform. Section A.2 presents an overview of the tools available on COW to launch and control parallel applications. Section A.3 briefly summarizes the key characteristics of the Tempest interface [Rei94]. Finally, Section A.4 presents an overview of Blizzard, discusses the default protocol libraries offered by Blizzard, and elaborates on the implementation of the Tempest primitives in Blizzard.

A.1 Wisconsin Cluster Of Workstations (COW)

Wisconsin COW consists of 40 dual-processor Sun SPARCStation 20s. Each contains two 66 Mhz Ross HyperSPARC processors [ROS93] with a 256 KB L2 cache memory and 64 MB of memory. The cache-coherent 50 Mhz MBus connects the processors and memory. I/O devices sit on the 25 Mhz SBus, which is connected through a bridge to the MBus. Each COW node contains a Vortex card [Pfi95] and a Myrinet network interface [BCF+95]. The Vortex card plugs into the MBus and performs fine-grain access control by snooping bus transactions. If the Vortex card is not present, two additional HyperSPARC processors can take its place, raising the total number of processors per node to four. Each node also contains a Myrinet

interface, which consists of a slow (7–8 MIPS) custom processor (LANai -2) and 128 KBytes of memory. The LANai performs limited protocol processing and schedules DMA transfers between the network and LANai memory or LANai memory and SPARC memory. The LANai processor cannot access SPARC memory directly. However, it can move data using DMA operations. Moreover, it can interrupt the host processor. The Myrinet switches are connected in a tree topology with the nodes attached as leaves of the tree.

A.2 Distributed Job Manager (DJM)

Jobs intended to run on the COW processing nodes are submitted through DJM [Cen93]. The version of DJM running on the COW is based on the CM-5 1.0 version. However, it has been heavily modified for the COW context by Mark Dionne [Dio96]. Readers interested in actually running DJM jobs are referred for more details to the DJM man pages from which the following information was extracted. We first overview the DJM commands for controlling DJM jobs and the application interface through which processes can take advantage of the DJM support for parallel applications.

DJM distinguishes two types of COW workstations: processing nodes and servers. The servers are used to run the controlling processes for a job. Jobs may be submitted from any workstation that has been authorized in the DJM configuration file. The following commands are used to control DJM jobs on COW.

crun/csub These commands are used to request execution of a COW job. The filename given on the command line will be executed on a server. *crun* will run a job interactively and display the output on screen, while *csub* will queue the job for batch execution, sending the output to a file. If the command does not specify a file to execute with crun, the user will be connected to a shell process on the server. The *crsh* command can then be used to execute commands on the allocated set of nodes.

cdel <jobid> This command will delete the DJM job from the queue, or kill it if it's currently running. Without parameters, it will delete the job owned by the issuer; if you have more than one job running or queued, you'll need to specify a job ID.

cstat It lists currently active jobs. This will print the user name, job ID, partition name, partition nodes, server name, total runtime used and requested, status, and command running on the server.

cstat proc It lists current processes running under the control of DJM on all nodes and servers. The job ID of the parent job is listed. With the -all flag, it lists all user-level processes running on all nodes and servers.

cstat up It lists the current status of the COW nodes and servers. NODE STATUS indicates whether a node is reachable, and EXECD STATUS indicates whether the DJM execd is running, making the node available for jobs.

cstat free It shows the current number of available nodes for each partition and the total number of available nodes. It also shows the number of "down" nodes, and the size of the largest available contiguous set.

The following environment variables are set in the server process, and will also be propagated to any process started via **crsh**:
DJM_JID: Job ID.
DJM_GID: Group ID allocated to the job's processes.
PART_SIZE: Number of allocated nodes.
PART_PROCNUM: Position of the current node in the partition.
PART_BASEPROC: Identifies the first node in the partition. This variable is only meaningful for contiguous sets of nodes.
PART_HOSTS: Colon separated list of nodes allocated to the job. (e.g., cow01:cow03)
CP_HOST: Address of the server. (as an IP string)
CP_PORT: Port on which the server is listening.

The following two functions that allow programs to interact with DJM, are provided by the cow library. Blizzard uses these functions to create an initial communication channel between the application processes until low-latency messaging has been initialized in all the nodes.

*int cow_register_pn(cow_pn_info_t *info, int open_socks);*
This function can be called by each process started on the processing nodes. Upon return, the info structure will contain information about the current job and its nodes. Each node opens a listening socket, connects to the control process on the server and sends it information about its own port number. The control process gathers information from all nodes, and then broadcasts a complete table of node names and ports back to them. If the *open_socks* flag is set, each node will connect to each other node in the job. The socket numbers will be returned in the info structure.

void cow_sync_with_nodes(void);
This function returns after all nodes have called it. The synchronization is done through the first node in the partition.

A.3 Tempest Interface

Tempest is an interface that enables user-level software to control a machine's memory management and communication facilities [Rei94]. Software can compose the primitive mechanisms to support either message-passing, shared-memory, or hybrid applications. Since these primitives are available to programs running at user level, a programmer or compiler can tailor memory semantics to a particular program or data structure, much as RISC processors enable compilers to tailor instruction sequences to a function call or data reference [Wul81]. Tempest provides four types of mechanisms:

Virtual Memory Management. With user-level virtual-memory management, a compiler or run-time system can manage a program's address space. Tempest provides memory mapping calls that coherence protocol uses to manage a conventional flat address space. In the context of a commodity operating system, this mechanism is trivially implemented using system calls that change the page protection for the shared heap.

Fine-Grain Access Control. Fine-grain access control is the Tempest mechanism that detects reads or writes to invalid blocks or writes to read-only blocks and traps to user-level code, which uses the resulting exceptions to execute a coherence protocol action [HLW95]. At each memory reference, the system must ensure that the referenced datum is local and accessible. Therefore, each memory reference is logically preceded by a check that has the following semantics:

```
if (!lookup(Address))
   CallHandler(Address, AccessType)
memory-reference(Address)
```

If the access is allowed, the reference proceeds normally. Otherwise, the shared-memory protocol software must be invoked. This sequence must execute atomically to ensure correctness. In the Tempest model, fine-grain access control is based on tagged memory blocks. Every memory block—an aligned, implementation-dependent, power-of-two size region of memory—has an access tag of *ReadWrite*, *ReadOnly*, or *Invalid*[1] that controls allowed accesses (see Table A.1). The actual block size supported by an implementation is referred to as the implementation's *minimum block size*, and blocks of the minimum block size are called *minimal blocks*. Invalid accesses trigger a block access fault. These faults suspend the thread making the access and invoke a user-level handler. A typical handler performs the actions dictated by a coherence protocol to make the access possible. Then, it updates the tag, and the access is retried.

Table A.1: Result of memory operations

Access	Tag		
	Invalid	**ReadOnly**	**Writable**
load	fault	return data	return data
store	fault	fault	write data

Low-Overhead Messaging. Low-overhead "active" messages [vECGS92] provide low-latency communication, which is fundamental to the performance of many parallel programs.

1. Tempest defines a fourth state, *Busy*, which has the same semantics as the *Invalid* state. This state is supported in Blizzard but no protocol library has ever used it.

In Tempest, a processor sends a message by specifying the destination node, handler address, and a string of arguments. The message's arrival at its destination creates a thread that runs the handler, which extracts the remainder of the message with *receive* calls. A handler executes atomically with respect to other handlers, to reduce synchronization overhead.

Bulk Data Transfers. Efficient transfers of large quantities of data is critical for many static computations. In Tempest, a processor initiates a bulk data transfer much like it would start a conventional DMA transaction, by specifying virtual addresses on both source and destination nodes.

A.4 Blizzard Implementation

This section starts with an overview of Blizzard. It continues to discuss the default coherence and synchronization protocol libraries offered by Blizzard. Finally, it elaborates on the Tempest primitives for virtual memory management, fine-grain access control, fine-grain messaging, bulk data transfers, and thread management focusing on their implementation in Blizzard.

A.4.1 Blizzard Overview

Blizzard applications use the "single program, multiple data" (SPMD) model. In this model, the same program text is loaded at the same location in every address space, though each processor executes that text independently. Each address space also contains other per-node private segments for data and stacks. Shared memory is supported only for regions allocated through special *malloc*-like calls that manage a shared heap, a contiguous segment at the same location in each address space that it is designated as the *shared memory segment*. Within this segment, the user handles accesses to unmapped pages, and controls the accessibility of mapped memory at a fine granularity.

Blizzard supports the PARMACS programming model [BBD+87]. PARMACS offers to each process of a parallel application a private address space with fork-like semantics while shared memory support is limited to the special shared memory segment. Blizzard preallocates an address range within the application address space for the shared memory segment. Initially, accesses to the shared region are disabled by setting the page protection to none for all the pages in the region. On the first access to a page in the shared heap, a fault occurs which the operating system forwards to a Blizzard signal handler. Tempest specifies user-level page fault handlers and exports primitives which toggle the page protection to read-write and enable file-grain access control for that page.

Each address space can support multiple threads of execution. These threads may execute concurrently on multiprocessor nodes. One of the threads is distinguished as the *protocol thread*. All other threads are referred to as *computation threads*. The protocol thread exists solely to execute user-defined *handler functions* to process *protocol events*: network message arrivals and the page faults and block access faults of computation threads. In the current

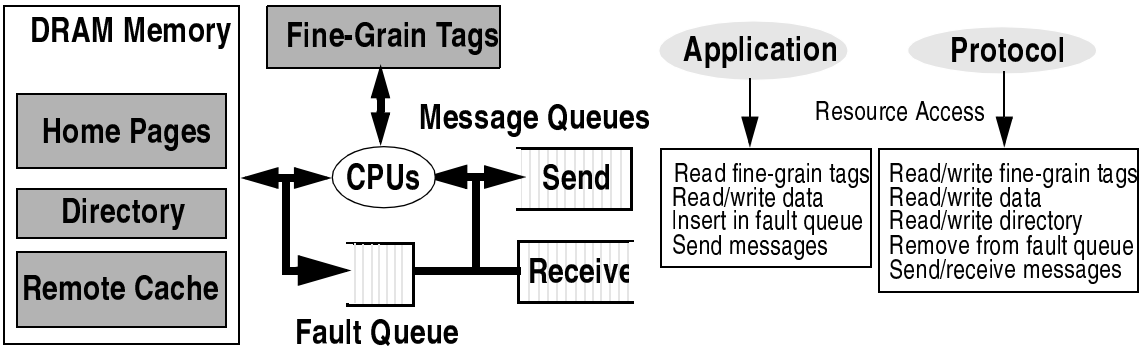

Figure A-1. Resources in a FGDSM system.

COW implementation, handler functions are executed sequentially but other synchronization models are also possible.

Figure A-1 illustrates the resources required to implement FGDSM systems. Shared data pages are distributed among the nodes, with every node serving as a designated *home* for a group of pages. A *directory* maintains sharing status for all the memory blocks on the home nodes. A *remote cache* serves as a temporary repository for data fetched from remote nodes. A set of *fine-grain tags* enforce access semantics for shared remote memory blocks. Upon access violation, the address of the faulting memory block and the type of access are inserted in the *fault queue*. Processor(s) are responsible for running both the application and the software coherence protocol that implements shared memory. The protocol moves memory blocks among the nodes using *send message queues* and *receive message queues.*

Figure A-1 also illustrates a breakdown of the resources accessed by the application and the protocol respectively. The application verifies access semantics upon a memory operation by reading the fine-grain tags. Memory operations also may read or write data to either home pages or the remote cache. An access violation in the application will insert an entry into the fault queue. The protocol manipulates the fine-grain tags to enforce access semantics to shared data, manages the remote cache data and state, maintains the list and status of sharers in the directory, removes entries from the fault queue, and sends and receives messages.

Blizzard supports four different fine-grain tag implementations. These designs cover a substantial range of the design space for fine-grain access control mechanisms [SFL+94]. Blizzard/S uses a software method based on executable-editing technology locally available [LS95]. Blizzard/E uses commodity hardware to accelerate the access check. The memory controller supports *Invalid* blocks while the TLB is used to emulate *ReadOnly* blocks. Blizzard/ES also uses the memory controller for *Invalid* blocks but relies on the same software method as Blizzard/S for *ReadOnly* blocks. Finally, Blizzard/T uses Vortex, a custom board [Pfi95] to accelerate fine-grain many access control operations.

Blizzard supports six different thread models. The *float-st* model supports a single computation thread and it is targeted to uniprocessor nodes. This implementation uses one processor and it relies on polling code inserted in the application executable to interleave protocol processing with the application execution. The *fix-st* model also supports a single computation thread running on a single processor. However, a dedicated processor performs all of the protocol processing. The *float* model supports an arbitrary number of processors. Each processor executes a single computation thread but all processor may do protocol processing. The *fix* model also supports an arbitrary number of processors with a single computation thread per processor. As in *fix-st,* all the protocol processing is performed on a dedicated protocol processor. The *float-mt* and *fix-mt* are similar to the *float* and *fix* models respectively but they are able to support more than one computation thread per processor.

Blizzard currently supports one messaging implementation. Blizzard's communication library started from the LANai Control Program (LCP) used in Berkeley's LAM library [CLMY96]. The LCP was modified to fit the Tempest active message model and to improve performance for its expected usage. The changes are small, and, in fact, the modified LCP is still source-level compatible with Berkeley's LAM library. An experimental messaging subsystem relies on System V shared memory to support messaging through shared memory hardware. This implementation has been used to identify scalability bottlenecks in the messaging subsystem when using multiprocessor nodes with a large number of processors. Initial Blizzard prototypes on COW have also used other implementations based on the Illinois Myrinet FM library or Unix sockets. This code however, has not been maintained is not currently functional.

Blizzard's source tree Assuming that the environment variable *$(WWT_ROOT)* points to the root of the source tree, the first level contains the *$(WWT_ROOT)/include*, *$(WWT_ROOT)/lib.src*, and *$(WWT_ROOT)/lib* directories. *$(WWT_ROOT)/include* contains the externally visible header files that protocol libraries and applications need to include. *$(WWT_ROOT)/lib.src* contains the source files for the Blizzard systems as well as standard protocol libraries that accompany the Blizzard distribution. Finally, *$(WWT_ROOT)/lib* is the directory where the compiled versions of the Blizzard system and applications libraries reside. The next level in both *$(WWT_ROOT)/include* and *$(WWT_ROOT)/lib.src* follows the same organization. The subdirectory *cow/blizzard* contains the Blizzard core files. The subdirectory *cow/sys* contains COW-specific files. The subdirectory *cyklos* contains the implementation of the thread package for Blizzard systems that support more than one computation thread. Finally, the subdirectory *prot* contains the default coherence and synchronization libraries distributed with Blizzard.

The files in the *$(WWT_ROOT)/lib.src/cow/blizzard/* source tree are organized in a way as to facilitate the code reuse among different Blizzard systems. *$(WWT_ROOT)/lib.src/cow/blizzard/common/* contains source files shared across all the Blizzard variants. *$(WWT_ROOT)/lib.src/cow/blizzard/$(ACCESS)/"* implements the corresponding fine-grain access control tags. *$(WWT_ROOT)/lib.src/cow/blizzard/$(NETWORK)/* contains the network code for a particular network implementation. *$(WWT_ROOT)/lib.src/cow/blizzard/*

$(MODEL)/$(ACCESS) contains the routines that are specific to a particular combination of fine-grain access implementation and thread model.

A.4.2 Blizzard Protocol libraries

Tempest's default coherence protocol is an 128-byte S-COMA-like [HSL94] software protocol [RLW94]. Stache maintains internal data structures to correctly implement a sequentially consistent shared memory coherence protocol. For each shared page, one node acts as the home node. By default home pages are distributed among the nodes in a round-robin manner. However, sophisticated programs are able to optimize the allocation using a first-touch policy. With this policy, all home pages are allocated on the first node during the initialization phase. After the parallel phase begins, any access to a page by a node, will result in that node being the home node for the page. This allocation policy results in improved performance because normally Stache is a four-hop protocol unless the home node is one of the sharer nodes. More specifically, the first time that some node other then the home node accesses a memory location in a page, it will fault. The Stache fault handler will request a copy of the block from the home node. If the block is not available at the home node, it will be forwarded to the node that has the most recent copy. That node will send the block back to the home. From there, it will be forwarded to the requesting node. Since there are four messages involved to service this request, the protocol is called a four-hop protocol. If however, no other nodes were involved only two messages need to be exchanged.

The *util* protocol library implements synchronization primitives, typically used in PARMACS applications. The primitives supported include locks and barriers. The user can choose at compile time the lock implementation between message and MCS [MCS91] locks. In message locks, the default lock implementation, the locks are distributed in a round-robin fashion among the nodes. Then, each lock or unlock request is implemented by sending an explicit message to the node that the lock resides. MCS locks are shared memory-based locks. Message locks perform better under high contention for the lock. Accordingly, MCS locks perform better when there is little or no contention. Barriers are implemented in a straightforward manner; all nodes send a message to node zero and the node responds when it collects the message from all the nodes. Better barrier implementations may be possible but they haven't been explored because barrier performance has not been critical to the applications we have examined.

A.4.3 Virtual Memory Management & Fine-Grain Access Control

Tempest gives user-level control over a region of the virtual address space on each node. This region is located at the same address in every process address space. Physical memory allocation and address translation are performed on the basis of pages. Pages mapped in the user-managed shared memory segment are referred to as *user-managed* pages. Only user-managed pages support fine-grain access tags. Each user-managed page must be assigned a *page mode number*, which determines the set of block access fault handlers that are invoked for block access faults on that page.

Two files, *blk_acc.c* and *pgtbl.c*, implement the Tempest virtual memory management and fine-grain access control primitives. Different versions of these files are maintained for the different fine-grain access control implementations. All implementations require additional support either in the form of kernel modules for hardware mechanisms (ECC and Vortex) or binary rewriting tools to insert software checks.

The kernel support required for hardware fine-grain access control has been designed using a layered approach. At the bottom layer, the Blizzard System Module encapsulates the hardware independent support. This module exports well-defined interfaces that hardware specific kernel modules can take advantage of these interfaces to expose the appropriate functionality to the user processes. The BLZSysM module consists of distinct components that collaborate to modify the behavior of Solaris to support the fine-grain memory abstraction. Two hardware specific modules have been implemented. ECCMemD and VortexD are the device drivers that support fine-grain access control using the ECC technique and the Vortex board, respectively. This layered design avoids code duplication since much of the functionality for both hardware-specific device drivers is the same. In addition, it facilitates the coexistence of two drivers simultaneously. For more information on the design and implementation of these modules, interested readers are directed to the main chapters of my thesis. The source files for all the kernel modules and drivers are located inside the Solaris 2.4 source tree under the *uts/sun4m* architecture-specific directory. Three separate directories, *blzsysm, eccmemd,* and *vortexd,* contain the source for the BlzSysM, EccMemD and VortexD modules, respectively.

The binary rewriting tool [SFH+96] used to insert the software checks for Blizzard/S is based on the EEL executable rewriting library [LS95]. The EEL library provides extensive infrastructure to analyze and modify executables. Besides software access control, I have been using this tool for rewriting Blizzard executables for other purposes as well (e.g., implicit network polling). The source files for the binary rewriting tool are located under the directory *$(WWT_ROOT)/tools/blizzard_s2.*

The following Tempest virtual memory management primitives have been implemented in Blizzard:

*int TPPI_alloc_and_map(void *pg, TPPI_PageMode mode, TPPI_BlkAccTag acc, TPPI_NodeId home, void *user_ptr);*
In Blizzard, this function initializes the fine-grain access control for a user-managed page. The page mode is used for block access fault handler selection and it is set according to *mode*. The block access tags for all blocks on the page are initialized to *acc*. The *home* and *user_ptr* fields are not interpreted by the system, but are intended to hold the node identifier of the page's directory node and a pointer to a per-page protocol data structure, respectively. The return value is zero if the call was unsuccessful. The call will fail if *pg* is not aligned to the page size, a mapping already exists for the virtual address *pg*, or *pg* is not within the user-managed virtual segment.

t*ypedef void (*TPPI_PageFaultHandlerPtr)(void *va, int pc, int is_write);*
void TPPI_register_page_fault_handler(TPPI_PageFaultHandlerPtr fn);
It registers *fn* as the user's page fault handler. When a computation thread accesses an unmapped virtual address in the user-managed virtual segment, Blizzard notifies the user by causing the protocol thread to invoke this function. The handler will be invoked as:

*void (*fn)(void *va, int pc, int is_write)*

where *va* is the unmapped address that was accessed and *pc* is the program counter of the load or store that caused the fault. The *is_write* parameter is non-zero if the access was a store, or zero if it was a load. The faulting thread is suspended until *TPPI_resume_va* is called, either by the page fault handler, a future message handler, or a different thread. Before resuming the faulting access, the page fault handler may directly request needed data from a remote node or it may simply initialize all blocks on the page to *TPPI_Blk_Invalid*. In the latter case, when the access is retried after the thread is resumed it will generate a block access fault. The page fault handler typically uses *TPPI_alloc_and_map* to add a page at the desired address. It may need to send a request to a remote node to determine the appropriate page mode; in this case, the *TPPI_alloc_and_map* and the resumption of the faulting thread will be performed by the response message handler.

*void *TPPI_get_page_user_ptr(void *va);*
*int TPPI_get_page_info(void *va, TPPI_PageInfo *info_ptr);*
*TPPI_PageMode TPPI_get_page_mode(void *va);*
*TPPI_NodeId TPPI_get_page_home(void *va);*
It provides information about the virtual page containing the arbitrarily-aligned address *va* in the user-managed virtual segment. The return value is 0 if the page is not mapped, 1 if it is mapped, and -1 if there is an error (because va is not in the user-managed virtual segment or info_ptr is not suitably aligned). If *info_ptr* is non-null and the return value is 1, the *mode, user_ptr,* and *home* values provided when the page was mapped are written to the fields of the same name in the structure pointed to by *info_ptr. TPPI_PageInfo* contains the *mode, user_ptr,* and *home* fields, which can also be retrieved individually by the other three functions.

*void TPPI_NodeId TPPI_get_page_prot(void *va);*
*void TPPI_NodeId TPPI_set_page_prot(void *va, TPPI_PageProt prot);*
These two function control the page protection for the user-managed page at address *va* once the fine-grain access control for a page has been enabled. They are used to implement the first-touch page allocation policy. Currently, only two page protection values, *TPPI_Page_Invalid* and *TPPI_Page_ReadWrite* are defined. Introducing a read-only value is significantly complicated, especially in Blizzard/E where the page protection is used to emulate the *ReadOnly* block access state. The functions will fail with an error message if *va* is not a user-managed page or the page protection is not one of two defined values.

*int TPPI_unmap_and_free(void *pg);*
*int TPPI_remap(void *old_pg, void *new_pg);*
TPPI_unmap_and_free removes the mapping for user-managed page pointed to by the

aligned address pg. *TPPI_remap* removes the mapping for user-managed page *old_pg* and remaps the physical page to the address *new_pg*, which must also be in the user-managed shared memory segment. The block access tags, page mode, home node ID, and user pointer are unchanged. Both *old_pg* and *new_pg* must be page-aligned. These two functions are currently not implemented in Blizzard. They were initially included in Tempest to allow flushing of remote pages cached in the local node to their home node. These experiments were conducted by Mike Litzkow on Blizzard/CM-5. However, the resulting experimental protocols were not included in the main source base and therefore, implementing these primitives was not high in my priority list. Note that hardware fine-grain tag implementations require the appropriate kernel support to implement these functions. Currently, such support has been implemented for the ECC technique (Blizzard/E, Blizzard/ES) but it has not yet been implemented for the Vortex board.

The following Tempest fine-grain access control primitives have been implemented in Blizzard:

```
typedef enum {
        TPPI_Blk_Validate_RW, TPPI_Blk_Upgrade_RW,
        TPPI_Blk_Validate_RO,TPPI_Blk_Downgrade_RO,
        TPPI_Blk_Invalidate, TPPI_Blk_Mark_Busy,
        TPPI_Blk_No_Tag_Change, TPPI_Blk_Invalid_To_Busy,
        TPPI_Blk_Busy_To_Invalid
} TPPI_BlkTagChange;
void TPPI_change_blk_acc(void *blk_va, int blk_len, TPPI_BlkTagChange chg);
void TPPI_change_blk_acc_and_get(void *blk_va, int blk_len,
TPPI_BlkTagChange chg, void *from);
void TPPI_change_blk_acc_and_put(void *blk_va, int blk_len,
TPPI_BlkTagChange chg, void *from);
```

Fine-grain tag modifications are made using the TPPI_BlkTagChange constants. These not only specify the desired tag value but also imply the current value of the tag. If the user applies a tag change operation to a block whose tag is not one of those implied by the operation the result is unspecified. All listed functions change the access tag of the block specified by *(blk_va, blk_len)*. In addition, *TPPI_change_blk_acc_and_get* and *TPPI_change_blk_acc_and_put* atomically change the tag of the block according to *chg* and copy data from or to the block specified by *(blk_va, blk_len)*. Threads may be executed concurrently on implementations with multiprocessor nodes, so while one thread is in the middle of a tag change, other threads may issue loads and stores. Operations that combine data transfer and access tag changes (including the messaging primitives *send_*Ba*, *send_*Bs*, and *recv_Ba* that will be discussed later) provide the following semantics:

- If data is read from a block and the block's access is downgraded from Writable, the block data that is read is guaranteed to reflect all writes that complete before the tag change.
- If data is written to a block and the block's access is upgraded, any load or store that does not fault but would have faulted given the previous access tag is guaranteed to be performed after the block's contents are updated with the new data.

*TPPI_BlkAccTag TPPI_get_blk_acc(void *va);*
Returns the block access tag associated with the block containing *va*. The result is undefined if the address is not in the user-managed segment or has not been not initialized.

void TPPI_register_blk_acc_fault_handle(TPPI_PageMode mode,
TPPI_BlkAccTag tag, int acc, TPPI_BlkAccFaultHandlerPtr fn);
This function registers function fn as the block access fault handler for accesses of type *acc* to blocks tagged with tag on pages of mode. The handler will be invoked as

*void (*fn)(void *va, void *user_ptr, TPPI_NodeId home)*

where *va* is an address within the block on which the faulting access was performed and *user_ptr* and *home* are the values supplied to *TPPI_alloc_and_map* when the page containing *va* was initialized.

A.4.4 Fine-Grain Messaging

Fine-grain messaging provides low-overhead messages, optimized for short message lengths. The first word of every message is the starting program counter of the handler to be executed at the receiver. Messages handlers are executed serially by the protocol thread. Tempest supports two variants of message primitives to initiate the transfer of small and large messages respectively. Primitives that send small messages accept only word arguments. Primitives that send large message accept a memory block specifier in addition to the word arguments. The memory block specifier can refer to a memory block (cache-block aligned) or a memory region (word aligned). The former are optimized for transferring cache blocks between nodes and among their arguments include the tag change for the memory block that should occur atomically with the message injection. The latter are used for more general message operations on non-aligned regions.

Originally, Tempest supported only receiver-based addressing, in which the handler invoked at the receiver end upon message arrival, could specify the destination for the incoming message data. To support zerocopy transfers (minimal messaging), Blizzard has been extended the Tempest interface with a second set of primitives that allows the destination virtual address to be specified when the message is sent. When the handler for messages sent through these primitives is invoked, the data have already been copied to the final destination.

Moreover, the original Tempest primitives only synchronous send semantics. In other words, the assumption is that after the operation is invoked the data area can be safely modified without affecting the outgoing message. To support such semantics with minimal messaging, the send primitive does not return until after the data have been pulled down to the NI. While this allows to support minimal messaging within the old interface, it also limits performance. To alleviate this problem, a new set of messaging primitives with asynchronous semantics has been introduced. These calls schedule messages and return immediately with a token. It is the responsibility of the user code to check that the messaging operation has been completed by invoking with this token an appropriate query primitive that informs the caller about the status of that message.

With Myrinet hardware, the host (SPARC) processor and Myrinet LANai processor cooperate to send and receive data. The division of labor is flexible since the LANai processor is programmable. However, the SPARC processor is far faster, which effectively limits the LANai processor to simple tasks [PLC95]. A Solaris device driver maps the Myrinet interface's local memory into a program's address space where it can accessed with uncached memory operations. To permit DMA between the LANai and the user address space, the driver allocates a kernel buffer mapped into the user address space. The LCP implements, in LANai memory, separate send and receive queues. Each queue contains 256 entries consisting of a message header and up to 8 words of data. The protocol supports small messages up to 8 words and large messages up to 4 Kbytes. The queues are fully addressable by the host processor since they are software structures in memory, implemented as circular buffers.The host processor uses loads and stores to move the word data—the complete message for small messages and the header for large messages—in and out of LANai memory. The LANai processor uses DMA to move larger messages through intermediate kernel/user buffers.

Blizzard messaging subsystem is implemented within the files located in the *$(WWT_ROOT/cow/blizzard/ucbam* directory *(ipc.c, ipc_sends.c ipc_inlines.h, network.c).* However, a large portion of the support code to implement messaging over Myrinet is located outside the Blizzard source tree, in its own source directory *(/s/myrinet-3.0).* Blizzard uses the slightly modified LCP from the Berkeley AM distribution. Moreover, Blizzard uses the initialization routines out the host-side AM library. The LCP binary and the AM library are located in files are located in the */s/myrinet-3.0/lib* directory while the source files are located in the */s/myrinet-3.0/src/sparc_sunOS/lam* directory. Blizzard also uses initialization routines out of the Myrinet core libraries that are located in the */s/myrinet-3.0/dev* directory. Finally, the Solaris driver is located in the */s/myrinet-3.0/src/sparc_solaris/driver* directory. There are two components in the driver. The *dlpi* component supports TCP/IP over Myrinet. It is required to initialize the device on boot-up but it is largely inactive thereafter. The *mmap* component enables user-level messaging through the Myrinet interface by supporting direct application access to the device memory.

The following Tempest fine-grain message sending primitives have been implemented in Blizzard:

t*ypedef void (*TPPI_MessageHandlerPtr)(TPPI_NodeId src, int size);*
void TPPI_send_typelist(TPPI_NodeId dest, TPPI_MessageHandlerPtr pc, arglist);
This set of functions sends a message to the specified node, where the message will be handled by executing code starting at the specified program counter. The body of the message is constructed using the specified (possibly empty) item list. In the current C binding, the item list specification is split: the types of the items in are encoded in a string that is part of the function name, while the parameters describing the items are part of the argument list. A given item may require more than one parameter. The following item types are available (with the type string given in parentheses):

- *Word (W).* A single machine word is sent. The corresponding parameter is the word value, of type *int*.

- *Block with access change (Ba).* The contents of a memory block are sent, and the memory block's access tag is modified. The corresponding parameters are the block specifier (void *blk_va, int blk_len) and the tag change (type TPPI_BlkTagChange).
- *Sender directed block with access change (Bs).* Exactly as the previous item type, but extended to include two extra parameters that specify the destination virtual address (type void *) and the destination tag change (type TPPI_BlkTagChange).
- *Region (R).* The contents of a region of memory are sent. The region must start on a word boundary and contain an integral number of words. The corresponding parameters are the region start address (type void *) and the region length in bytes (type int).
- *Sender directed region (Rs).* Exactly as the previous item type, but extended to include an extra parameter that specifies the destination virtual address (type void *).
- *Forward (F).* Data from the current received message is sent. This option is only valid when the send is called in the context of a message handler. The corresponding parameter is the number of bytes to forward (type int), which must be a multiple of the word size.

On the receiver, the system logically queues the message until the protocol thread is idle. The sender-specified function is invoked with two parameters: the source node (type TPPI_NodeId) and the size of the message body in bytes (type int). The message body can be consumed using the calls listed below, which correspond to the types available for sending. In addition, message data can be consumed using a "forwarded block" item in a send operation. Unlike the Tempest specification, in Blizzard, the types used to send and receive a particular message are expected to match. It is relatively trivial to relax this restriction, but it results in worse performance and no protocol library or application currently rely on being able to do so. The following Tempest message reception primitives have been implemented in Blizzard:

int TPPI_recv_W();
The next word argument is returned.

*void TPPI_recv_Ba(void *blk_va, int blk_len, TPPI_BlkTagChange chg);*
The next *blk_len* bytes are read from the queue and written to the memory block specified by *(blk_va, blk_len)*, whose access tag is changed. This is a null operation for sender-addressed blocks.

*void TPPI_recv_R(void *va, int len);*
The next *len* bytes of data are read and written to the specified region of memory. The region must start on a word boundary and contain an integral number of words. This is a null operation for sender-addressed blocks.

A.4.5 Bulk data transfer

Bulk data transfer provides high-bandwidth, connection-oriented, memory-to-memory data movement between nodes. The interface has been roughly modeled from operating system interfaces used to manage direct-memory-access (DMA) controllers. The processors initiate a bulk data transfer much like it would start a conventional DMA transaction, by specifying vir-

tual addresses on both source and destination nodes. The channel interface was first developed for the Blizzard/CM-5 [SFL+94]. On the CM-5 platform, the small network packet size (5 words) supported by the network hardware, did not allow the efficient implementation of active messages with large data blocks. Therefore, the cost of breaking a bulk data transfer into small active messages was too high to realize bandwidth comparable to the one achieved by the default CM-5 messaging library [Thi91]. On the COW implementation, the use of the channel interface has been depreciated mainly because the fine-grain messaging layer can support messages up to 4 Kbytes, which are big enough to offer the maximum throughput possible in this platform. The channel interface is still supported by Blizzard, but it is implemented on top of fine-grain messaging layer.

A.4.6 Thread management & Protocol Dispatching

The protocol dispatch loop checks for protocol events such as access faults and incoming messages and launches the appropriate handlers. In Blizzard, the handler launch is done inside a critical section since Tempest dictates atomic handlers. There are different versions of the dispatch loop depending on the thread model (i.e., whether the Blizzard implementation supports multiple processors and threads and whether there exists a dedicated processor for message processing).

When a dedicated protocol processor is used, it repeatedly polls for access faults and messages and executes the respective handlers. It looks for messages in the receiver-overflow queue, in the local queue for messages self-addressed messages or in the LANai queue. Messages in the receiver overflow queue have higher priority that any other messages so that messages are delivered in-order. While in-order delivery is not required by Tempest, experience has shown that it makes the development of user protocols easier.

With implicit message polling on flow control back-edges using executable editing, two versions of the dispatch loop are required. The first is invoked when a message is detected from the polling code inserted in the executable. The second is invoked when the user protocol waits for a message. In this case, message processing must stop as soon as the waiting condition is satisfied to ensure forward progress and avoid livelocks.

The protocol dispatch code is located in one file, *proto.c*, particular to each combination of fine-grain access control implementation and thread model. The thread management code is located in two files, *wait_signal.c*, *fault_restore.s*, and *tppi_thread.c* that are shared among all Blizzard implementations. The following Tempest functions are implemented in these files:

*void TPPI_resume_va(void *blk_va, int blk_len);*
Resumes the set of threads suspended due to a page fault or block access fault on a user-managed page. In both cases, the faulting instruction is reissued.

*void TPPI_sleep(volatile int *sem_ptr);*
Increments the semaphore pointed to by *sem_ptr*. If the resulting semaphore value is greater

than zero, the calling thread is suspended until the semaphore value is less than or equal to zero.

*void TPPI_wakeup(volatile int *sem_ptr);*
Decrements the semaphore pointed to by *sem_ptr*. If the resulting semaphore value is equal to zero, any threads waiting on the semaphore (via *TPPI_sleep*) will be resumed.

*void TPPI_atomic_incr(volatile int *sem_ptr);*
Increments the semaphore pointed to by *sem_ptr*. Because Blizzard systems may schedule threads concurrently or preemptively, it is unsafe for users to perform read-modify-write operations directly on semaphores.

void TPPI_spin(SpinFuncPtr f, int arg);
void TPPI_shm_spin(SpinFuncPtr f, int arg);
Both functions block the calling thread while spinning until the value returned by the supplied function f invoked with argument arg is zero. In TPPI_spin, the function is restricted to accessing only local memory addresses (i.e., no access faults should occur in the function). In TPPI_shm_spin, the function can access shared memory addresses but the function has higher overheads and therefore, it is less efficient.

A.4.7 Compiling Blizzard Applications & Libraries

The directory *$(WWT_ROOT)/include/cow/blizzard* contains makefiles to facilitate the compilation of Blizzard applications and protocol libraries. The main makefile used to compile an application or protocol library must first set appropriate variables to select the Blizzard system and then it must include these makefiles.

Figure A-2 presents the Blizzard makefile used by a sample application. The makefile sets the *MODEL, ACCESS, NETWORK, BLKSZ, THREAD_PACKAGE, PROTOCOL, UTILS,* and *TARGET* variables to select the thread model, the fine-grain tag implementation, the messaging implementation, the coherence block size, the thread package implementation, the protocol library, the synchronization library, and the name of the target executable, respectively. The application source file dependencies follow and the makefile ends by including the standard application makefile from the Blizzard include directory.

The process to compile protocol libraries is similar to compiling Blizzard applications. Figure A-3 presents a makefile that it used to compile a sample protocol library. Again, the makefile sets the appropriate variables to select the Blizzard implementation and then it includes the standard library makefile from the Blizzard include directory.

A.4.8 Running Blizzard Applications

Launching Blizzard applications is done using the DJM commands discussed in Section A.2. The application processes, which are launched in this way on the COW nodes,

```
MODEL = floatst

ACCESS = es

NETWORK = ucbam

BLKSZ = 128

THREAD_PACKAGE = cyklosmqueue

PROTOCOL = stacheft$(BLKSZ)

UTILS = utils$(BLKSZ)

TARGET = barnes

CFLAGS += -g -O3

OBJS = code.o load.o grav.o

code.o: code.c code.h util.c

grav.o: grav.c code.h

load.o: load.c code.h

code.h: defs.h
```

Figure A-2. Sample Application Makefile.

initialize the system, establish the communication channels, and start executing the application code. As part of the initialization sequence, the command line arguments are processed. Blizzard scans the command line processing any arguments that it recognizes until it reaches an argument that it cannot process. The rest of the command line is passed to the application. The following list describes the most important Blizzard command line options.

-num-cpus (-ncpu) <int>:
: Sets the number of processors per node.

-num-threads (-nt) <int>:
: Sets the number of computation threads per node. With *fix*st* thread models, it must be equal to number of processors minus one. With *float*st* thread models, it must be equal to the number of processors. With **mt* thread models, it can be of any value up to the maximum number of threads supported by the thread

```
MODEL = floatst

ACCESS = es

NETWORK = ucbam

BLKSZ = 128

OBJS = home.o stache.o vm.o ni_init.o prot_stats.o \
       update.o stache_utils.o par_begin.o
```

Figure A-3. Sample Library Makefile.

package.

-myrinet-options (-bm) <hex>:

Sets options for myrinet.

The following values are used to enable minimal messaging:

0x1: Enable minimal messaging on sender

0x2: Enable minimal messaging on receiver

0x4: Minimal messaging on the receiver does not cause interrupts. Rather the system polls for failed translations.

The following values are used to select the message notification technique:

0x08: Set the poll location using interrupts.

0x10: Set the poll location using DMA.

0x20: Set the Vortex message bit, which is used as the poll location.

0x40:Use SIGUSR1 as the message notification mechanism.

-file-label (-la) <string>:

Sets the label for the output and statistic files.

-output-to-file (-o):

Sends the output to a file.

-system-time (-s):

Reports system(blizzard) time

-print-trace (-pt):

Sends the debug trace to file instead of the trace buffer.

-print-trace-buffer (-pb):

Sends the trace buffer to file when it becomes full. This is more efficient than *-pt* since I/O occurs only when the buffer becomes full.

-trace (-tr) <hex>:

Sets the debug tracing mask that selects the events to trace.

-trace-size (-ts) <int>:

Sets the number of trace buffer entries.

-loop-on-error (-le):

The system loops on error instead of exiting.

References

[AB86] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, June 1990.

[ACP94] Tom Anderson, David Culler, and David Patterson. A case for networks of workstations: NOW. Technical report, Computer Science Division (EECS), University of California at Berkeley, July 1994.

[ACP95] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for Networks of Workstations: NOW. *IEEE Micro*, February 1995.

[ADC97] Andrea C. Arpaci-Dusseau and David E. Culler. Extending proportional-share scheduling to a network of workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, June 1997.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–44, March 1983.

[BBD+87] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfieldand Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston Inc., 1987.

[BBLS91] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[BBO+83] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D . J. States, S. Swamintathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculation. *Journal of Computational Chemistry*, 4(187), 1983.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[BCL+95] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz. Remote queues: Exposing message queues or optimization and atomicity. In *Proceedings of the Sixth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 42–53, 1995.

[BCM94] Eric Barton, James Cownie, and Moray McLaren. Message passing on the Meiko CS-2. *Parallel Computing*, 20:497–507, 1994.

[BCZ90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 168–176, February 1990.

[BDFL96] Matthias A. Blumrich, Cesary Dubnicki, Edward W. Felten, and Kai Li. Protected User-level DMA for the SHRIMP Network Interface. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[Bel96] Gordon Bell. 1995 Observations on Supercomputing Alternatives: Did the MPP Bandwagon Lead to a Cul-de-Sac? *Communications of the ACM*, 39(3):11–15, March 1996.

[BGvN46] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument, 1946. Report to the U.S. Army Ordinance Department.

[BHMW94] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Proceedings of Supercomputing '94*, pages 590–599, November 1994.

[BJM+96] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Second USENIX Symposium on Operating Systems Design and Implementation*, October 1996. Seattle, WA.

[BLA+94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[BM95] Doug Burger and Sanjay Mehta. Parallelizing Appbt for a Shared-Memory Multiprocessor. Technical Report 1286, Computer Sciences Department, University of Wisconsin–Madison, September 1995.

[BS96] Jose Carlos Brustoloni and Peter Steenkiste. Effects of buffering semantics on I/O performance. In *Second USENIX Symposium on Operating Systems Design and Implementation*, October 1996. Seattle, WA.

[BS98] Manjunath Bangalore and Anand Sivasubramaniam. Remote subpaging across a fast network. In *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC)*, February 1998.

[BTK90] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: A language for distributed programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[BWvE97] Anindya Basu, Matt Welsh, and Thorsten von Eicken. Incorporating memory management into user-level network interfaces. In *Hot Interconnects '97*, 1997.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[CCBS95] Frederic T. Chong, Shamik D. Charma, Eric A. Brewer, and Joel Saltz. Multiprocessor runtime support for fine-grained, irregular DAGs. *Parallel Processing Letters: Special Issue on Partitioning and Scheduling*, December 1995.

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

[CDK+94] Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[Cen93] Army High Performance Computing Research Center. *Distributed Job Manager Man Pages*, 1993.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. Limitless directories: A scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[CL96] Satish Chandra and James R. Larus. HPF on Fine-Grain Distributed Shared Memory: Early Experience. In Utpal Banerjee, Alexandru Nicolau, David Gelernter, and David Padua, editors, *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*. August 1996.

[CLMY96] David Culler, Lok Tin Liu, Richard Martin, and Chad Yoshikawa. LogP performance assessment of fast network interfaces. *IEEE Micro*, pages 35–43, February 1996.

[CM88] Albert Chang and Mark F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[CMSB91] Eric Cooper, Onat Menziolcioglu, Robert Sansom, and Francois Bitz. Host interface design for ATM LANs. In *Proceedings of the 16th Conference on Local Computer Networks*, pages 14–17, October 1991.

[Cor94] Convex Computer Corp. *SPP1000 Systems Overview*, 1994.

[CRD+95] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teeodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 12–25, December 1995.

[CRL96] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[DAC96] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1996.

[DAPP93] Peter Druschel, Mark B. Abbot, Michael A. Pagels, and Larry L. Peterson. Network subsystem design. *IEEE Network*, 7(4):8–19, July 1993.

[DCF+89] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Nuth, Scott Wills, Paul Carrick, and Greg Flyer. The j-machine: A fine-grain concurrent computer. In G. X. Ritter, editor, *Proc. Information Processing 89*. Elsevier North-Holland, Inc., 1989.

[Dio96] Mark Dionne. *DJM-COW User Guide*, 1996. http://www.cs.wisc.edu/ cow/djm.html.

[DP93] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 189–202, December 1993. Asheville, NC.

[DPD94] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the ACM SIGCOMM '94 Conference*, pages 2–13, September 1994. London, UK.

[DR97] Dave Dunning and Greg Regnier. The virtual interface architecture. In *Hot Interconnects '97*, 1997.

[DW89] William J. Dally and D. Scott Wills. Universal mechanism for concurrency. In *PARLE '89: Parallel Architectures and Languages Europe*. Springer-Verlag, June 1989.

[DWB+93] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, pages 36–43, July 1993.

[EK89] Susan J. Eggers and Randy H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 257–270, 1989.

[ENCH96] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. Softflash: Analyzing the performance of clustered distributed virtual shared memory supporting fine-grain shared memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[Fal97] Babak Falsafi. *Fine-Grain protocol execution mechanisms and scheduling policies on SMP clusters*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, 1997.

[FLR+94] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[For94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Draft, University of Tennessee, Knoxville, May 1994.

[FW96] Babak Falsafi and David A. Wood. When does dedicated protocol processing make sense? Technical Report 1302, Computer Sciences Department, University of Wisconsin–Madison, February 1996.

[FW97] Babak Falsafi and David A. Wood. Scheduling communication on an SMP node parallel machine. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 128–138, February 1997.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994.

[GCP96] R. Gillett, M. Collins, and D. Pimm. Overview of Memory Channel Network for PCI. In *Proceedings of the 41th IEEE Computer Society International Conference (COMPCON'96)*, 1996.

[Gil94] George Gilder. The bandwidth tidal wave. *Forbes ASAP*, December 1994. Available from http://www.forbes.com/asap/gilder/telecosm10a.htm.

[GJ91] David B. Gustavson and David V. James, editors. *SCI: Scalable Coherent Interface: Logical, Physical and Cache Coherence Specifications*, volume P1596/D2.00 18Nov91. IEEE, November 1991. Draft 2.00 for Recirculation to the Balloting Body.

[Gro95] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.1*, 1995.

[Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HGDG94] John Heinlein, Kourosh Gharachorloo, Scott A. Dresser, and Anoop Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 38–50, 1994.

[HHS+95] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[Hil87] Mark Donald Hill. Aspects of cache memory and instruction buffer performance. Technical Report UCB/CSD 87/381, Computer Science Division (EECS), University of California at Berkeley, November 1987. Ph.D. dissertation.

[Hil97] Mark D. Hill. Multiprocessors should support simple memory consistency models. Technical Report CSD TR97-1353, Department of Computer Science, University of Washington, October 1997.

[HLRW93] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in it Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V).

[HLW95] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A substrate for portable parallel programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[Hor95] Robert W. Horst. TNet: A reliable system area network. *IEEE Micro*, February 1995.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[HSL94] Erik Hagersten, Ashley Saulsbury, and Anders Landin. Simple COMA node implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, January 1994.

[Hsu94] Peter Yan-Tek Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[Inc91] Sun Microsystems Inc. *SPARC MBus Interface Specification*, April 1991.

[Int90] Intel Corporation. *iPSC/2 and iPSC/860 User's Guide*. Intel Corporation, 1990.

[Int93] Intel Corporation. Paragon technical summary. Intel Supercomputer Systems Division, 1993.

[Int96] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual*, January 1996.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 213–228, December 1995.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[KA93] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the alewife multiprocessor. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, 1993.

[KBG97] Alain Kagi, Doug Burger, and James R. Goodman. Efficient synchronization: Let them eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[KBM+96] Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris MC: A multi computer OS. In *Proceedings of the 1996 USENIX Conference*, page ?, January 1996.

[KDCZ93] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. Technical Report 93-214, Department of Computer Science, Rice University, November 1993.

[Ken92] Kendall Square Research. Kendall square research technical summary, 1992.

[KHS+97] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michal Cierniak, Srinivasan Parthasarathy, Jr. Wagner Meira, Sandhya Dwarkadas, and Michael Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[kJC96] Hsiao keng Jerry Chu. Zero-copy tcp in solaris. In *Proceedings of the '96 USENIX Conference*, January 1996. San Diego, CA.

[KMRS88] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, (3):79–119, 1988.

[KS96] Magnus Karlsson and Per Stenstrom. Performance evaluation of a cluster-based multiprocessor build from ATM switches and bus-based multiprocessor servers. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[K+94] Jeffrey Kuskin et al. The stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software Engineering*, 13(1):23–31, January 1987.

[Lab97] Ames Scalable Computing Laboratory. Microsoft Wolfpack references. http://www.scl.ameslab.gov/workshops/wolfpack.html, February 1997.

[LB94] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software Practice & Experience*, 24(2):197–218, February 1994.

[LC95] Lok T. Liu and David E. Culler. Evaluation of the Intel Paragon on active message communication. In *Proceedings of Intel Supercomputer Users Group Conference*, 1995.

[LC96] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA computer for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LRV94] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory system support for parallel language implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 208–218, San Jose, California, 1994.

[LS95] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[LW94] Alvin R. Lebeck and David A. Wood. Fast-cache: A new abstraction for memory system simulation. Technical Report 1211, Computer Sciences Department, University of Wisconsin–Madison, January 1994.

[Mar94] Richard P. Martin. Hpam: An active message layer for a network of HP workstations. In *Hot Interconnects '94*, 1994.

[MC95] Alan Mainwaring and David Culler. *Active Messages: Organization and Applications Programming Interface*, 1995.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[MFHW96] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[MGBK96] Christine Morin, Alain Gefflaut, Michel Banatre, and Anne-Marie Kermarrec. Coma: an opportunity for building fault-tolerant scalable shared memory multiprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[Mic91] Sun MicroSystems. *Sun-4M System Architecture, Rev 2.0*, 1991.

[Mic94] Sun MicroSystems. *Kodiak SX Memory Controller Specification*, 1994.

[MK96] Evangelos P. Markatos and Manolis G.H. Katevenis. Telegraphos: High-performance networking for parallel processing on workstation clusters. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, 1996.

[MKAK94] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. Fugu: Implementing translation and protection in a multiuser, multimodel multiprocessor. Technical Memo MIT/LCS/TM-503, MIT Laboratory for Computer Science, October 1994.

[Mog91] Jeff Mogul. Network locality at the scale of processes. In *Proceedings of the ACM SIGCOMM '91 Conference*, September 1991. Zurich.

[MP97] Christine Morin and Isabelle Puaut. A survey of recoverable distributed shared virtual memory systems. *IEEE Transactions on Parallel and Distributed Systems*, September 1997.

[MRF+97] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[MSH+95] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[MVCA97] Richard P. Martin, Annin M. Vadhat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[Nik94] Rishiyur S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Languages and Compilers for Parallel Computing (Proceedings of the Sixth Internationa Workshop)*, pages 390–405. Springer-Verlag, 1994.

[NIM93] NIMBUS Technology. NIM 6133 memory controller specification. Technical report, NIMBUS Technology, 1993.

[NMP+93] A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Borwne, G. Aybay, and D. Lee. S3.mp: A multiprocessor in a matchbox. In *Proc. PASA*, 1993.

[OG90] R. R. Oehler and R. D. Groves. IBM RISC system/6000 processor architecture. *IBM Journal of Research and Development*, 34(1):32–36, January 1990.

[Ope92] Open Software Foundation and Carnegie Mellon University. *Mach 3 Kernel Interfaces*, November 1992.

[Osb94] Randy Osborne. A hybrid deposit model for low overhead communication in high speed lans. In *Fourth International Workshop on Protocols for High Speed Networks*, August 1994.

[OSS80] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[OZH+96] Randy Osborne, Qin Zheng, John Howard, Ross Casley, and Doug Hahn. DART - a low overhead ATM network interface chip. In *Hot Interconnects*, 1996.

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, May 1990.

[PD97] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Perspective*. Prentice Hall, 1997.

[Pfi95] Robert W. Pfile. Typhoon-Zero implementation: The Vortex module. Technical report, Computer Sciences Department, University of Wisconsin–Madison, 1995.

[PKC97] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processor. *IEEE Concurrency (to appear)*, 1997.

[PLC95] Scott Pakin, Mario Laura, and Andrew Chien. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[RAC96] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High performance local area communication with fast sockets. In *Proceedings of the '96 USENIX Conference*, January 1996. San Diego, CA.

[Rei94] Steven K. Reinhardt. *Tempest Interface Specification, Rev 1.1*, 1994.

[Rei96] Steven K. Reinhardt. *Mechanisms for Distributed Shared Memory*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, December 1996.

[RFW93] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[RHL+93] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[ROS93] ROSS Technology, Inc. *SPARC RISC User's Guide: hyperSPARC Edition*, September 1993.

[RPW96] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[SCB93] Daniel Stodolsky, J. Brad Chen, and Brian Bershad. Fast interrupt priority management in operating systems. In *Second USENIX Symposium on Microkernels and Other Kernel Archtitectures*, pages 105–110, September 1993. San Diego, CA.

[Sco96] Steve L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 26–36, 1996.

[SFH+96] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lucas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, and David A. Wood. Implementing fine-grain distributed shared memory on commodity smp workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[SFH+97] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, and David A. Wood. Blizzard: Cost-effective fine-grain distributed shared memory. Submitted for Publication, 1997.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[SGA97] Daniel J. Scales, Kourosh Gharachorloo, and Anshu Aggarwal. Fine-grain software distributed shared memory on smp clusters. Technical Report 97/3, Digital Equipment Corporation, Western Research Laboratory, February 1997.

[SGC93] Rafael H. Saavedra, R. Stockton Gaines, and Michael J. Carlton. Micro benchmark analysis of the KSR1. In *Proceedings of Supercomputing '93*, pages 202–213, November 1993.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[SPE90] SPEC. SPEC benchmark suite release 1.0, Winter 1990.

[SPWC97] Patrick G. Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic coscheduling on workstation clusters. Submitted for publication, March 1997.

[SS86] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[Ste94] Peter Steenkiste. A systematic approach to host interface design for high-speed networks. *IEEE Computer*, March 1994.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[TH90] V. Tam and M. Hsu. Fast recovery in distributed shared virtual memory systems. In *IEEE 10-‘th International Conference on Distributed Computing Systems*, 1990.

[Thi91] Thinking Machines Corporation. The connection machine CM-5 technical summary, 1991.

[TJ92] Yuval Tamir and G. Janakiraman. Hierarchical coherency management for shared virtual memory multicomputers. *Journal of Parallel and Distributed Computing*, 15(4):408–419, August 1992.

[TL94a] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 110–119, 1994.

[TL94b] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, California, 1994.

[TLL94] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating data and control transfer in distributed operation systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, California, 1994.

[UNMS94] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart Sechrest. Tapeworm II: A new method for measuring os effects on memory architecture performance. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 132–144, October 1994.

[Vah96] Uresh Vahalia. *Unix Internals: The New Frontiers*. Prentice Hall, 1996.

[vE93] Thorsten von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. PhD thesis, UCB, November 1993.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 40–53, December 1995.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrating communication and computa-

tion. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[WF90] K.L. Wu and W.K. Fuchs. Recoverable distributed shared memory. *IEEE Transactions on Computers*, pages 460–469, April 1990.

[WG94] David L. Weaver and Tom Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[WGH+97] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu, Thomas Wicki, and Winfried Wilcke. The Mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[WH95] David A. Wood and Mark D. Hill. Cost-effective parallel computing. *IEEE Computer*, 28(2):69–72, February 1995.

[WHJ+95] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages; a mechanisms for scheduling communication with computation. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 217–226, August 1995.

[Wil92] John Wilkes. Hamlyn — an interface for sender-based communications. Technical Report HP-OSR-92-13, HP, November 1992.

[Woo96] David A. Wood. Personal communication, October 1996.

[Woo97] David A. Wood. Personal communication, December 1997.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.

[Wul81] William A Wulf. Compilers and computer architecture. *IEEE Computer*, 14(7):41–47, July 1981.

[YKA96] Donald Yeung, John Kubiatowicz, and Anant Agarwal. MGS: A multigrain shared memory system. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[ZIS+97] Yuanyuan Zhou, Liviu Iftode, Jaswinder Pal Singh, Kai Li, Brian R. Toonen, Ioannis Schoinas, Mark D. Hill, and David A. Wood. Relaxed consistency and coherence granularity in dsm systems: A performance evaluation. In *Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, June 1997.