An Efficient Virtual Network Interface in the FUGU Scalable Workstation

by

Kenneth Martin Mackenzie

S.B., Massachusetts Institute of Technology (1990)
S.M., Massachusetts Institute of Technology (1990)

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

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1998

Author . . . .
Department of Electrical Engineering and Computer Science
22 December 1997

Certified by . . . .
Anant Agarwal
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
M. Frans Kaashoek
Associate Professor of Computer Science and Engineering
Thesis Supervisor

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

An Efficient Virtual Network Interface in the FUGU Scalable Workstation
by
Kenneth Martin Mackenzie

Submitted to the Department of Electrical Engineering and Computer Science on 22 December 1997 in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY

Abstract

A scalable workstation is one vision of a mainstream parallel computer: a machine that combines scalable, fine-grain communication facilities for parallel applications with virtual memory and preemptive multiprogramming to support general-purpose workloads. A key challenge in a scalable workstation is the Virtual Network Interface (VNI) problem. The problem is that high performance communication for parallel programming depends on a tight coupling between the application and the network while multiprogramming and virtual memory effects disrupt such coupling.

This thesis introduces and evaluates the "direct" virtual network interface: a solution to the VNI problem for fine-grain messages in a scalable workstation. The direct VNI employs two complementary architectural techniques to reconcile speed and protection. First, *two-case delivery* optimistically provides direct, user-level access to network interface hardware but also transparently backs the direct system with a robust, software-buffered system. Two-case delivery allows the scalable workstation to support both good parallel application performance through the fast hardware interface and good global system performance by permitting buffering when required for multiprogramming. Second, the software-buffered mode uses *virtual buffering* to provide effectively unlimited buffer capacity by storing messages in dynamically managed virtual memory. Virtual buffering gives the user the convenient illusion of a very large buffer while giving the operating system the means to minimize actual, physical memory consumption.

The direct VNI ideas are implemented in an experimental scalable workstation, FUGU, consisting of emulated hardware, a matching simulator and a custom operating system. Results from workloads of real and synthetic applications show that the direct VNI provides high performance because the direct case is both fast and common. Microbenchmarks show the protected direct delivery case costs only 60% (10s of cycles per message) more than unprotected messages on the same hardware. Further, in a mixed workload experiment, we observe that our parallel applications see only $14 - 33\%$ of messages buffered when 10% of the CPU time is devoted to unrelated, high-priority, interactive tasks. Finally, results show that physical buffering requirements remain naturally low in real applications despite the combination of unacknowledged messages and unlimited buffering.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Engineering

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

My experience at MIT has been an enjoyable one from essentially the moment I arrived, so it is with a distinct sense of regret that I finally submit this document and bid MIT goodbye. I have been privileged to work in an environment that is simultaneously so challenging and so supportive.

A number of specific acknowledgements are due to people who administered the challenges and the support directly. First, all my colleagues and friends in the Alewife group have been invaluable and will be sorely missed. The two-case message delivery approach was inspired by John Kubiatowicz' "network overflow" mechanism in Alewife and many of the details were worked out in subsequent conversations with Kubi. Kirk Johnson and Matt Frank each at different times served to provide incisive "scientific" advice, most welcome general discussions and fabulous backfield coverage on the ice. Don Yeung, across the desk, provided useful feedback, support and a quiet sense of competition which I look forward to continuing to enjoy. Anne McCarthy's competence kept group operation blissfully smooth.

My advisors provided crucial enthusiasm and energy that I can only hope to be able to emulate. Anant's unfailing optimism and high-level vision directly fostered the creative work environment of the Alewife group. Frans provided steady feedback and served as an energetic role model. Larry provided a key external viewpoint and refreshing perspective.

Several people contributed to the FUGU system quite directly. Rob Bedichek put together a novel simulator that was fast, useful and, most of all, easy for us to understand and to extend. Jon Michelson designed and implemented the TLB for FUGU and did most of the FPGA work. Victor Lee designed and implemented the second network and studied its role in deadlock avoidance and overflow control. Victor also built the FUGU node board, with help from Eric Bovell and Silvina Hanono. Walter Lee built a fine parallel scheduler for FUGU and endured enless suggestions from myself, Matt and Larry.

Finally, I am indebted to my parents whose encouragement led me to MIT and to Lori whose encouragement helped me stay.

This research was supported in part by NSF grant # MIP-9012773, in part by ARPA contract # N00014-94-1-0985, in part by a NSF Presidential Young Investigator Award to Anant Agarwal and in part by a NSF National Young Investigator Award to M. Frans Kaashoek.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The design of a parallel computer to be used as a general-purpose machine must include both efficient, scalable communication facilities and support for standard, multiuser operating system features. The problem is that high performance communication for parallel programming depends on a tight coupling between the application and the network while the effects of operating system features of multiprogramming and virtual memory disrupt such coupling. Consequently, most existing systems either do not address protection for multiple users or virtualize the communication system only at the expense of either limited performance or of restrictions to the programming model.

This thesis introduces and evaluates the *direct virtual network interface* (direct VNI): a set of techniques for providing communication in the form of fine-grain message passing in a way that is both efficient and protected. The application controls a hardware network interface directly when possible while operating system software transparently synthesizes a virtual interface by buffering messages in memory when (rarely) required. Like virtual memory, the direct VNI gives an application performance near that achievable on the bare hardware while giving the operating system the ability to multiplex and manage limited resources.

The direct VNI ideas are implemented in an experimental multiprocessor, FUGU, consisting of emulated hardware, a matching simulator and a custom operating system. Results from workloads of real and synthetic applications show that the direct VNI provides high performance because applications commonly use the fast interface in hardware. Microbenchmarks show that message delivery via the protected hardware interface costs only 60% (10s of cycles per message) more than unprotected messages on the same hardware. Using a mixed workload experiment, we observe that our parallel applications see at most $14 - 33\%$ of messages buffered when 10% of the CPU time is devoted to unrelated, high-priority, interactive tasks. Finally, results show that physical memory requirements remain naturally low in real applications despite the convenience of a programming model that effectively guarantees delivery of messages.

The remainder of this chapter defines and further motivates the problem domain, briefly introduces the direct VNI solution and then summarizes the contributions of the thesis. Section 1.1 introduces the *scalable workstation*, our vision of a general-purpose parallel computer, Section 1.2 summarizes the challenges arising in a scalable workstation, including the key virtual network interface problem, Section 1.3 describes the direct VNI and Section 1.4 summarizes contributions.

Approach	Efficient Communication	Scalable Interconnect	Multiuser Features
SMPs	×		×
Clusters		×	×
MPPs	×	×	
Scalable Workstations	×	×	×

Table 1-1. Qualitative comparison of machines in terms of support for efficient communication, support for scalable communication and support for multiuser operation. SMPs, clusters and MPPs each exhibit some of the desired qualities.

1.1 A Scalable Workstation

Parallel processors are currently widely used but have yet to become ubiquitous. A *scalable workstation* is one view of the result of evolution in general-purpose parallel computers: a machine that combines high performance, scalable communication mechanisms with support for protection. Existing machines exhibit a subset of the desirable features of good communication performance, scalable performance and protected, multiuser operation.

Parallel processors have become commonplace as high-end workstations, as clusters and as large, dedicated machines. Small-scale symmetric multiprocessors (SMPs) are used as high-end workstations or as servers. Such machines are expected to run a mixed workload of interactive, workstation-like applications as well as parallel applications. Work on larger parallel applications is increasingly focussed on making use of clusters of workstation-class uniprocessors or SMP-based multiprocessors connected with commodity local-area network (LAN) components. Clusters are again expected to support a mixed workload of parallel, interactive applications, such as world-wide web search services or data mining services, as well as parallel, compute-intensive applications such as scientific and engineering simulation models. A few commercial, massively-parallel processors (MPPs) exist running primarily parallel, scientific codes.

An ideal, general-purpose multiprocessor design would be capable of high performance with mixed workloads and good cost/performance over a range of machine sizes. Existing SMPs, clusters and MPPs each fall short of some aspect of this goal. Table 1-1 summarizes the characteristics of each type in the space of communication performance (crucial to parallel performance), scalability (communication performance that scales with with the number of processors) and support for standard multiuser operating system features. Each of the existing machine types exhibits some but not all of the desired qualities of a general-purpose multiprocessor. We discuss each of the rows in Table 1-1 below.

Current SMPs work well as small, general-purpose multiprocessors but are limited by the use of a broadcast bus (or other broadcast medium) as the interprocessor interconnect. The bus structure makes communication efficient and, in particular, makes cache-coherent shared memory easy to support. However, the bus forms a global bottleneck that limits the scale of SMP machine sizes to a few nodes. The solution is to replace the bus with a point-to-point network and to synthesize shared memory communication by using hardware or software to communicate via messages over this network. Hardware solutions using directory-based cache coherence have been demonstrated in research prototypes [46, 1, 41] and high-end commercial machines [36, 42, 49, 79]. prototypes [48, 10, 35, 30, 65, 67, 37, 21] using a variety of techniques. Future solutions are likely to make use of judicious combinations of hardware and software support in hybrid machines that support both shared memory and message passing communication [62, 1].

Clusters of workstations or SMPs interconnected by a LAN provide support for multiple users and can provide scalable cost and performance, but current clusters provide only inefficient inter-node communication. Clusters typically support only message-passing for inter-node communication in hardware and synthesize shared memory (if desired) in software. The bottleneck in clusters is the interface to the interconnection network and sometimes the network itself. Network interfaces (NIs) for cluster nodes have evolved according to the demands of local area networks, *i.e.,* using heavyweight protocols that tolerate uncertain network characteristics and the view that the network is a peripheral device. Current work in cluster NIs seeks to reduce protocol overhead, to tighten the integration of the NI with the processors and to take advantage of the characteristics of the so-called System Area Network (SAN) environment [77, 64, 81, 6, 19, 17, 29, 13].

Higher performance network interfaces suitable for significantly finer-grain parallel problems have been demonstrated in massively-parallel processors as research prototypes [70, 7, 16, 1, 61, 2, 56] and as commercial machines [45, 69, 72]. However, MPP work has largely ignored issues of mixed workloads that require multiprogramming, demand paging and interactive scheduling.

A scalable workstation represents one vision of the convergence of SMP, cluster and MPP goals and technologies that combines efficient communication, a scalable interconnect and multiuser support. Specifically, a scalable workstation has the following characteristics:

- It will run a mixed workload, consisting of both interactive, response-time sensitive jobs and compute- and communication-intensive parallel jobs.
- It will support efficient communication including both explicit communication through message passing and implicit communication through shared memory.
- It will use a scalable, point-to-point interconnect with the characteristics of a SAN, *e.g.,* high reliability, effective hardware flow control and low latency.

Scalable workstations are expected to run a mix of the workloads that desktop workstations and SMP servers run now and to enable the use of parallel programs for more applications. Scalable workstations fit naturally into either a "workstation" or a "network computer" model of organization. In a workstation model, scalable workstations serve as desktop computers tailored in size to the needs of the user. In a network-computer model, desktop machines are minimally functional and are backed by group-level or departmental-level scalable workstations as servers tailored in size to the needs of the group. Despite partial centralization, the network-computer model remains workstation-like if each desktop head is guaranteed the resources of some number of processors in the backing server.

A scalable workstation supports a full set of communication primitives. Shared memory is widely considered more easily programmable than message passing. Message passing remains desirable as well for several reasons. First, the strengths of message-passing for bulk transfer and explicit synchronization are complementary to the strengths of shared memory for automatic communication [38], making a mixed model attractive. Second, raw message-passing can have performance benefits over shared memory in reduced interconnect traffic and more robust latency

	Multiprogramming	Virtual Memory
Message Passing	Isolation Parallel scheduling	Page faults in handlers DMA coherence Translation for DMA
Shared Memory	*(Isolate using VM)* Parallel scheduling	Translation Coherence

Table 1-2. Challenges arising from the integration of four features desired in a scalable workstation. Message passing and shared memory are features desired for high performance parallel processing while multiprogramming and virtual memory are features desired for general-purpose, multiuser operation.

tolerance [12]. In particular, even if written assuming a shared-memory programming model, a program in which communication patterns are amenable to compile-time analysis might make better use of compiler-generated message passing communication than of even hardware-supported shared memory [14, 12]. Third, distributed shared memory and message passing implementations are naturally similar at a low level, so if an implementation provides shared memory in hardware the additional cost of exposing message passing is low [40]. Finally, one reading of technology trends is that shared memory systems increasingly will be implemented in software using messages to minimize hardware [1, 31] and/or to take advantage of application-specific knowledge [62].

Finally, a scalable workstation is made "scalable" by its point-to-point interconnect which allows the communication performance of the machine to scale with the number of processors in the machine. Scalability has two consequences. First, the global *bandwidth* of the network potentially scales up as processors are added. Scaling network bandwidth with the number of processors allows good application speedups for larger machines and for a larger class of applications than a system with a fixed network bandwidth. Second, the *cost* of the interconnect scales with the number of processors. The benefit is that a single machine implementation can exhibit good cost/performance for a wide range of machine sizes including small sizes. In contrast, for instance, the bus in an SMP is cost effective for only a narrow range of machine sizes.

A scalable workstation is an attractive vision. However, the combination of mixed workloads, mixed communication models and the distributed nature of the system leads to a number of challenges, described next.

1.2 Challenges in a Scalable Workstation

The primary challenges in a scalable workstation arise from the integration of communication features with multiuser features. Message passing and shared memory are the two communication models. Multiprogramming and virtual memory are features arising from the goal of supporting a mixed, multiuser workload. Table 1-2 enumerates the challenges that arise from integrating these features by considering the intersection of each of the communication models with each of the multiuser features.

There are a number of important challenges, only the first of which is pursued in detail this thesis. The discussion below explains each challenge listed in the table and regroups them into four named problems.

- First is the "Virtual Network Interface" (VNI) problem which encompasses the top row of items in Table 1-2. Messages in a multiuser system must be isolated from one another and resilient to disruptions such as page faults in virtual memory.[1] At the same time, the interface must allow for efficient communication. Research on communication mechanisms supports that view that efficiency comes from tight coupling of the network with the application. The demands of isolation tend to interfere with tight coupling from both directions. For protection reasons, the network may not be able to deliver a message immediately. Similarly, due to multiprogramming, an application may not be able to *receive* a message immediately. Demand-paged virtual memory causes a similar effects: a page fault (or a remote shared memory miss) in message handling code introduces a delay in message reception that may be intolerable to the network. We proposed a solution to the VNI problem in [50] and partly evaluated it in [51]. This VNI solution is implemented in FUGU and is the focus of this thesis. Other recent network interface work addresses the VNI problem with similar goals, notably CNI [58], the *T family [61, 2] and the M-machine [25]. These projects are described as related work in Chapter 8.

- Second is the DMA problem. Efficient bulk transfer through messages requires the support of Direct Memory Access (DMA) hardware or equivalent functionality provided by a coprocessor. Using DMA with virtual memory requires virtual address translation for the DMA engine. Either the DMA engine itself must be capable of performing such translations or there must be a secure means for the DMA engine to receive physical addresses from the processor. Further, combining DMA with virtual memory (or with shared memory) introduces a data coherence problem because the DMA engine becomes an additional source of memory operations. We proposed a solution to the DMA problem in [50] and some elements of that solution are implemented in FUGU (Appendix A). Others have addressed DMA with virtual memory in network interfaces as well [80, 68].

- Third is the translation coherence problem. Virtual memory combined with shared memory introduces coherence problem with cached translations because virtual-to-physical mappings are conventionally cached at the processors. A solution to translation coherence needn't be as efficient as data cache coherence but must be scalable. We proposed a scalable solution to translation coherence in [50]. Teller examined a number of solutions to translation coherence [75].

- Finally, there is the scheduling problem. Parallel schedulers for mixed workloads are not yet fully understood. It appears important to be able to schedule some applications with traditional, per-processor, priority-based scheduling and others with coordinated scheduling ("coscheduling"). Network scheduling in Table 1-2 refers to the problem of controlling the impact of one application's network traffic on another. Traffic effects can be ameliorated by communication models that minimize blocking such as sender-based messages. Ultimately,

[1]The integration of shared memory with message passing presents problems that are similar to those presented by virtual memory. For instance, a cache miss to a remote shared memory location is similar to a page fault and could be handled in the same way.

however, network effects must be considered by a parallel scheduler. A "flexible coscheduling" parallel scheduler was proposed in [44] and is partly implemented in FUGU. Others are working on the same problem [22, 73].

While DMA, translation coherence and scheduling have each been studied to some extent in the FUGU system, the focus of this document is on the solution to the VNI problem.

1.3 An Efficient Virtual Network Interface

Existing solutions to the VNI problem compromise in one of three ways: in *performance*, by reducing the performance of the implementation, in *protection*, by limiting the support for multiprogramming, or in *programmability*, by limiting the flexibility of the communication model and therefore limiting its efficiency as a programming target.

This thesis presents the "direct" virtual network interface as a solution to the VNI problem that meets all three parts of the challenge. The direct VNI supports an aggressive, low-level message model, User Direct Messages (UDM), in which user messages logically correspond one-to-one with one-way, unacknowledged messages sent through the network. Messages are isolated for protection by tagging each message with a global identifier corresponding to the communicating application and by interpreting the tags appropriately.

The direct VNI architecture reconciles efficiency with protection by recognizing that protection failures are uncommon. The thesis introduces two complementary techniques as part of the architecture. First, the direct VNI uses *two-case delivery* to optimistically provide direct, user-level access to network interface hardware with a transparent, software-buffered fallback delivery system. The result is a system that simultaneously supports both good parallel application performance through an efficient, low-level interface and enables good global system performance through flexible multiprogramming. Second, the software-buffered mode uses *virtual buffering* to provide effectively guaranteed message delivery while giving the operating system the freedom to automatically manage physical buffering resources.

The two-case delivery and virtual buffering ideas are evaluated using workloads of real and synthetic applications running on a simulator and partly on emulated hardware. The results answer two main questions. First, results show the direct path is also the common path under most conditions, justifying the use of software buffering. Experiments show that only $14 - 33\%$ of messages in our sample parallel applications take the buffered path when 10% of CPU time is devoted to uncorrelated interactive tasks on 16 processors. Second, further results show that physical buffering requirements remain low (a few pages) for our sample applications despite the combination of unacknowledged messages and unlimited buffering.

The combination of UDM with an implementation based on two-case delivery and virtual buffering makes for a network interface that is both efficient and virtualized and that is particularly appropriate for the environment of a scalable workstation. The ideas are usable separately.

1.4 Contributions

The FUGU project has been a cooperative effort involving a number of people making overlapping contributions. The particular contributions of this thesis are fourfold:

1. We identify the virtual network interface problem and enumerate its major issues.
2. We present the direct virtual network interface, which applies the architectural techniques of two-case delivery and virtual buffering with overflow control to the virtual network interface problem. The thesis includes a detailed description of the two techniques. The essence of the UDM programming model and the specific hardware used to virtualize user interrupts have also been described by Kubiatowicz in the context of the Alewife machine [40].
3. We report on the implementation of a scalable workstation, FUGU, that uses the direct virtual network interface. FUGU consists of emulation-based hardware, a companion instruction-level simulator and a custom operating system. Features of the hardware have been previously described by Michelson [55] and by Lee [43].
4. We present an evaluation of the direct virtual network interface based on simulation and emulated hardware. The evaluation makes three basic points. First, the direct VNI in FUGU has best-case receive-side overhead near that of an unvirtualized interface built on the same hardware (within 60% or 10s of cycles). Second, the direct case is in fact the common case; under realistic mixed workload conditions our parallel applications see only $14 - 33\%$ of messages buffered with 10% of CPU time is devoted to unrelated, interactive tasks. Finally, we observe that memory consumption remains naturally low in reasonable applications for a broad definition of reasonable and is controllable in other applications.

1.5 Roadmap

The rest of the document is organized as follows. Chapter 2 describes the aspects of the VNI problem and concludes with the thesis problem statement. Chapter 3 presents our direct VNI solution, including the programmer-visible model and an overview of the architectural approach. Chapters 4 and 5 describe and discuss the details of the two-case delivery and virtual buffering architectural techniques, respectively. Chapter 6 introduces the experimental FUGU system used to evaluate the direct VNI ideas. Chapter 7 describes the experimental evaluation and presents the results from application workloads run on the FUGU system. Chapter 8 summarizes related work and Chapter 9 concludes.

Chapter 2

The Virtual Network Interface Problem

A key issue in a scalable workstation is the problem of reconciling efficient communication with standard support for multiprogramming. Efficient communication implies a tight coupling between application code and the communication system: the processes running an application can rely on timely action by the communication system and vice versa. Multiprogramming and related support, notably demand-paged virtual memory, prevent a process in an application from being able to guarantee its own timely behavior. We call the problem of reconciling communication efficiency with multiuser support the *virtual network interface* problem.

A full solution to the problem must address three major issues:

- Programmability: the interface to the communication system must serve as an efficient target for a programmer, a compiler or a runtime system. Assuming a good implementation, an interface model is good when it expresses naturally what the user needs to do and when it exposes the fundamental underlying costs of communication.
- Protection: the communication system must provide protection for multiprogramming and must be compatible with virtual memory and scheduling for mixed workloads.
- Performance: the communication system must transport data with low latency and high bandwidth. Ideally, given a particular programming model, the protected communication system will perform as well as raw, unprotected hardware.

Each of these three issues in isolation is not difficult. Supporting a programmable, protected interface while providing high performance is the real challenge. This chapter discusses each of the three issues and attempts to define an "ideal" for each against which any system should be compared. We will refer back to these points in the rest of the thesis when discussing the solutions in the direct VNI and in related work.

2.1 Programmability

The first issue is that the communication model must be both general and efficient. In the realm of fine-grain message passing models, low-level models that expose fundamental costs are attractive

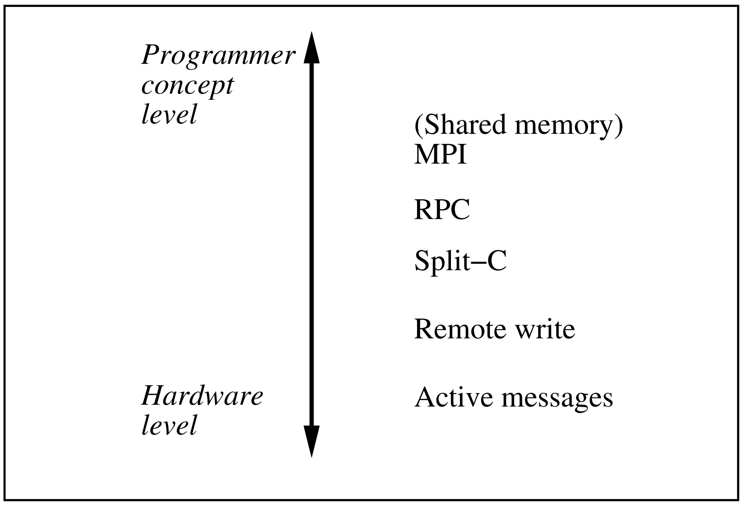


Figure 2-1. Existing communication models take a variety of approaches to bridging the programmer's notion of communication with the capabilities of the hardware. Here, an arbitrarily chosen set of models are arrayed on the right along an axis of "close to the program" at the top to "close to the hardware" at the bottom.

because they give the programmer the means to minimize those costs. We describe the desired functionality of a low-level model and enumerate the sources of costs.

The communication model of most interest in this thesis is the model for fine-grain messages. As described in the introduction, a scalable workstation is expected to support both shared memory and message passing communication models. Shared memory integrates naturally with virtual memory and multiprogramming so we consider only message-passing issues here. Further, message passing has two main purposes with different requirements: small, synchronizing messages and large, bulk transfer messages [38]. Fine-grain messages for combined data transfer and synchronization require low overhead and latency for maximum utility. Messages for bulk transfer demand chiefly high bandwidth because latency and per-message overhead are amortized over the time of the transfer. Thus, while low-overhead bulk transfer extends the usefulness of a bulk transfer mechanism, overhead is a secondary issue. We consider only small messages as the focus of this thesis and assume that bulk transfer messages are supported separately. Low-overhead bulk transfer support in FUGU is implemented as an extension of the direct VNI mechanism and is described in Appendix A.

Message-passing models for small messages are varied and controversial because, like an instruction set, a message model must balance naturalness as a programmer's abstraction with implementability in hardware. The model must serve as an efficient target for the application programmer, compiler or library writer. Simultaneously, the model must be implementable with high speed and using acceptable amounts of hardware. The tension between expressiveness and implementability leads to a tradeoff. Figure 2-1 sketches the tradeoff and loosely places some existing models in the space of the tradeoff. For instance, MPI [54] is a relatively high-level model. MPI defines synchronous send/receive operations and multicast/reduction operations as primitives. These primitives correspond to multiple messages in the network hardware. Close to the other extreme, Active Messages [78] is a low-level model with primitives that correspond closely to fundamental hardware operations. Each user message in Active Messages corresponds to a single hardware message through the network which invokes a user handler at the destination.

While a final conclusion awaits further research, we believe that the most successful interfaces in the long run are likely to be low-level ones that expose the fundamental costs of communication. In analogy to RISC instruction sets, a low-level abstraction both allows the hardware to be focused on support for a few, simple, fast primitives and gives software the opportunity to synthesize compound operations in an application-specific manner. At a minimum, a low-level model such as Active Messages can efficiently emulate a higher-level model such as MPI given a sufficiently low-overhead implementation. Efficiency in this context means that the emulation overhead of an Active Message implementation of, for instance, send/receive in MPI over a native implementation is negligible compared to fundamental costs such as data transfer time. More important than emulation, a low-level model offers the promise that network traffic and protocol overhead may be reduced over that required by a higher-level model by programmer specialization [78] or through automatic, compile-time analysis and specialization [23, 34].

An ideal low-level model provides a complete set of communication operations and exposes fundamental costs. The programmer is thus given the ability to craft communication protocols tailored to the application and to minimize communication costs using application-specific knowledge. The fundamental communication operations for fine-grain messages passing are data transfer and control transfer:

- Ideal data transfer moves data words from the right place to the right place, *i.e.,* from register/cache/memory of the source process to the register/cache/memory of the destination process, depending on the needs of the source and the destination. Small messages are likely to be transferred from register to register on the assumption that they are intended to be sent as soon as they are generated and to be consumed as soon as they are received.

- Ideal control transfer allows the sender and receiver to synchronize in the most convenient way; by having the receiver poll for a message, receive an interrupt when a message arrives, or with no synchronization at all. Both polling and interrupts have advantages and disadvantages in performance and in programmability. In terms of performance, interrupts work well for moderately frequent but latency-critical messages. Polling works well for predictable, high-volume message patterns or for latency-tolerant applications. In terms of programmability, interrupts give predictable performance without tuning. This is an advantage because periodic polling is often difficult to synthesize accurately. With polling, the atomicity model is clearer and less error-prone: a faulty polling-based application tends to deadlock, which is much easier to debug than a synchronization failure. [8].

The fundamental costs of communication arise from the operations of data and control transfer. Costs per message are due to latency and resource consumption in the network, in the endpoint (the network interface and/or the processor) and in any memory used if messages are buffering:

- Messages consume network bandwidth and suffer latency due to the network. Network resource consumption is a per-message cost.

- Message handling at the sending and receiving endpoints consumes computation bandwidth in the processor or network interface. Such computation also adds latency to the message.

Endpoint resource consumption is a per-message cost in a low-level model like Active Messages. A more complex model may use hardware or a coprocessor to reduce the endpoint cost for messages that are part of the complex model's protocol.

- Messages that are buffered in memory at some point consume memory storage space and consume message system bandwidth. Memory consumption is a per-message cost when buffering is used.

In addition:

- A message system incurs additional, possibly substantial, costs in the network, the endpoints and memory if a layer of protocol is required to synthesize message reliability, ordering, flow control, etc. [33], needed by the application.

The ideal communication model is both natural to program and is effective at minimizing the costs of communication. Low-level models work by exposing the fundamental operations and the fundamental costs based on the assumption that a compiler, a library implementation or the application programmer can best minimize the costs. In a lightweight, asynchronous model, each logical message specified by the programmer corresponds one-to-one with a message through the physical interconnect. This correspondence give the programmer maximum control over the traffic generated by the application and thus control over the per-message costs. Having each message invoke a handler on the processor gives the programmer the "glue" to build arbitrary protocols that minimize the required messages.

We have presented desirable programmability features and enumerated costs for fine-grain message-passing. The fine-grain messages mechanism is assumed to be complemented by support for bulk-transfer messages and for shared memory. The programmability of the virtual network interface in a scalable workstation is tempered by the need to support protection, described next.

2.2 Protection

The second issue in a virtual network interface is that the architecture must be compatible with multiuser operation in a scalable workstation. Multiuser operation requires protection between applications in all elements of the system: in the processors, in memory and in the network. A virtual network interface architecture must provide the protection for the network and must tolerate the *effects* of protection in the other elements of the system. The primary effect of protection from the point of view of the message system is that protection mechanisms can render network messages *undeliverable*, temporarily or permanently. This section discusses the undeliverability problem, protection for the network in particular and then protection for memory and the processor in a general way.

Undeliverable Messages. The primary consequence of the various aspects of protection from the point of view of the message system is that a message is not always deliverable to the destination process at the time a sender wishes to send. A message may be undeliverable due to application choice if the programmer's model allows a receiver to refuse messages. Multiuser protection features exacerbate the problem by introducing new reasons for messages to be undeliverable, *i.e.,* because

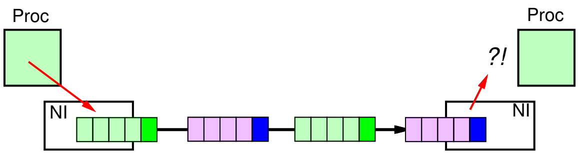


Figure 2-2. Protection requirements can make messages undeliverable. Here an incoming message from a different application has been mixed in with a parallel application's messages. The currently running (light colored) application cannot receive the (dark colored) message at the head of the queue for protection reasons. If the architecture permits potentially undeliverable messages to be launched into the network, the receiver must buffer, NACK or drop such messages.

the wrong process is scheduled at the receiver or because the receiving process takes a page fault. Figure 2-2 illustrates a situation in which a message has been launched from the sender on the left but the receiver is not ready to receive. How the receiver responds is a major architectural issue in a virtual network interface.

The focus of the discussion here is on protection rather than virtualization (naming) because it is protection that ultimately causes the undeliverability effect. Virtualization features, such as a mechanism to translate a virtual processor number in a message headers into a physical processor number, are useful, but protection is fundamental.

Network Protection. In the abstract, protection means preventing one application from discovering the results of another application or altering the execution of another application. Applied to the message system, protection means preventing an application from reading, forging or destroying another application's messages. More subtly, an application also must be prevented from unduly *delaying* messages belonging to another application. The effect of protection in the network is that messages may be made undeliverable at times due to protection conflicts.

Rigid scheduling strategies in a multiprocessor, *i.e.,* space partitioning or strict gang scheduling, can effectively solve the network protection problem by permitting only one application's messages in a portion of the network at a time. Rigid scheduling solves both the problems of isolation of data and of delay effects between individual messages. The CM-5 [45], for instance, provides protected multiprogramming by strict gang scheduling. Application messages are isolated because only one application is active in a partition at a time. The CM-5 network interface allows an application process to block the network, but the CM-5 limits the delay effect of such blocking to other applications by saving and restoring network state at application switch times. However, requiring rigid scheduling at all times is undesirable for interactive performance reasons.

Flexible scheduling implies that the messages of multiple applications may be active in the network at the same time. The network architecture must solve the two problems of isolating messages of multiple applications and of controlling the delay of one application on another application. Both problems lead to undeliverable messages.

The first part of network protection is isolating network data. With multiple applications sending messages simultaneously, isolation of network data requires tagging each message indelibly with information that allows it to be identified to a protection system which can in turn determine which process or processes may have access to the message. Tagging requires three mechanisms: the sending network interface must include a mechanism to apply the tag in a trusted way, the message in transit must encode the tag indelibly and the receiving network interface must include a mechanism to interpret the tag in a trusted way. The flexibility and expressiveness of the protection mechanisms may be arbitrarily elaborate, but at the bottom level the consequences are simple: messages must be identifiable while in transit and certain messages may be undeliverable due to protection conflict.

The second part of network protection is minimizing the delay effect of one application on another. Some amount of interaction between applications is inevitable since the applications share resources. However, for instance, if one application refuses to receive its own messages, perhaps because it polls only infrequently, the result can be a network blockage. The network interface must receive messages and keep network traffic moving whether the application is ready to receive messages or not. This problem is another instance of undeliverability.

Memory and Processor Protection. Memory and processor protection are conventional in uniprocessor workstations and largely extend naturally into scalable workstations. From the point of view of the network interface, memory and processor protection are simply other potential sources of undeliverability: a message can become undeliverable because it requires a memory or processor resource that is temporarily or permanently unavailable. For instance, a message handler that takes a page fault on its first instruction renders that message undeliverable for the (presumably intolerable) period required to service the fault. Parallel scheduling is a whole topic unto itself. We discuss it here in the context of message passing because scheduling has an impact on what the message system can expect.

A scalable workstation requires a system scheduler that caters to needs of both interactive, response-time-sensitive applications and parallel, synchronization-intensive applications. A system with an interactive workload needs a scheduler that provides good response time to interactive events. Standard, priority-based scheduling addresses this requirement well [74]. The natural extension of a priority-based scheduler to a multiprocessor is to schedule processes independently on each processor of the multiprocessor. However, parallel jobs, particularly ones that perform inter-process synchronization frequently, often require some form of coscheduling for best performance [60]. The conflict in scheduler requirements is stark in a scalable workstation where we want to run mixed workloads. A standard solution does not yet exist although a number of researchers are working on the problem [22, 73, 44]. Essentially what is required is a scheduler that works well in both interactive- and coscheduled modes.

Beyond the scheduler, the virtual network interface itself needs to operate efficiently in both modes. Coscheduled applications can easily make use of the low latency of a direct interface since coscheduling means that the sender and receiver of a message tend to be scheduled simultaneously. Applications running under standard, priority-based scheduling may be able to make use of a direct interface sometimes or "optimistically" but in general will require buffering of messages that arrive when the wrong process is scheduled.

The ideal support for protection in a virtual network interface provides isolation between individual messages so that multiple applications may be scheduled independently on each processor if application characteristics do not demand coscheduling. Ideally, the message system isolates the virtual networks of applications both in terms of correctness and in terms of performance and imposes no additional overhead in either the independently-scheduled or coscheduled scenarios.

2.3 Performance

Finally, a virtual network interface architecture must address the issue of performance given the demands of programmability and protection. The challenge of programmability with a low-level model is to provide full functionality with overheads near hardware limits. The complication introduced by protection is the problem of undeliverable messages. This section discusses the challenges of a low-level model and of undeliverability from the perspective of performance, then concludes by comparing the characteristics of two means of building a network interface in hardware: direct and buffered. A direct interface which brings network queues all the way to the processor offers low overhead for the operations in a low-level model. A buffered interface which queues messages in memory can solve the undeliverability problem. We conclude that a successful approach will combine the characteristics of each.

Programmability. A low-level model, as described in Section 2.1, demands full functionality and addresses communication costs by relying on the compiler, library or programmer to minimize messages. The performance challenge in the architecture and implementation is minimizing the per-message costs.

A particularly difficult case is the situation of message delivery via interrupt. There are two overheads associated with delivery via interrupt: the cost of having the processor take the interrupt and the delay of transporting the first word of data from the network interface to the processor.

A low-level model is most useful if the network provides reliable, exactly-once message transport semantics and flow control so that low-level operations can be used without an additional layer of protocol-level fault tolerance. A point-to-point network in one cabinet (a System Area Network or SAN) may be made to be sufficiently reliable that protocol-level fault tolerance becomes unattractive compared to a combination of link-level fault tolerance (*e.g.,* via ECC) and simple end-to-end fault detection. Other researchers take the same view [9, 25]. MPP manufacturers have found it feasible to build reliable networks for machines with several hundreds of nodes [69]. Placing the network in one cabinet avoids most of the practical causes of failures, *e.g.,* those due to unpluggings, independent power supply failures, cable damage and electromagnetic interference.

Protection. As discussed above, a consequence of the various aspects of protection is that messages are not always deliverable to the destination at the time a sender wishes to send. Messages may be undeliverable even in a single-user system due to application choice if the programmer's model allows a receiver to refuse messages. Multiuser protection features exacerbate the problem by introducing new reasons for messages to be undeliverable, *e.g.,* because the wrong process is scheduled at the receiver. There are a small number of possible solutions to undeliverability. Of these, buffering is attractive because, in the spirit of a low-level model, it works without adding the requirement of additional protocol messages.

There are four options for dealing with undeliverable messages, one sender-side and three receiver-side. The first option is to rule out undeliverable messages by prenegotiating all resources needed for a message transfer before a message is launched. The second, third and fourth options are to drop, to negatively acknowledge (NACK) or to buffer such messages.

Prenegotiation is widely used. For instance, with a remote-write model, the system can require that the sender can only name locations at the destination that are known to physically exist. As another instance, with a synchronous send/receive model, the receive statement can be used to trigger

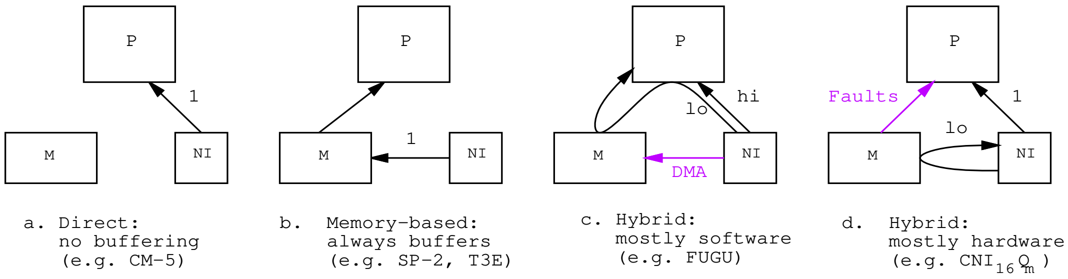


Figure 2-3. A network interface my delivery messages directly, may buffer all messages in memory or may take a hybrid approach. The annotations on the arcs represent relative frequencies along each path.

the send. Prenegotiation has costs in extra protocol overhead and potentially in delay. Prenegotiation is most useful when its costs are amortized over a bulk transfer or when the idea of prenegotiation is propagated all the way back to the model, *e.g.,* a shared memory or remote memory model, so that the programmer can work around its costs.

Of the three receiver-side options, dropping is the least desirable because it introduces unreliability into a system that may otherwise have adequately reliable hardware. Dealing with unreliability adds the extra costs of buffering copies at the sender and of extra traffic to manage the copies.

NACKs introduce complexity into the network interface, add traffic and overhead for buffer management and require a reserved back-path. The "return-to-sender" strategy, used in the T3E [69] and the M-machine [25] is a form of NACK that avoids much of the common-case buffer overhead. Return-to-sender assumes a reliable network. Both dropping and NACKing have the effect of increasing network demand under load. Further, dropping and NACKing are subject to livelock unless additional steps are taken to prevent it.

Finally, buffering has the advantage of letting messages always make forward progress, at least with respect to traversing the network. However, buffering only postpones the undeliverability problem unless the buffer is effectively infinite in size or the total buffer space required by the workload is provably limited by some means. The SP-2 takes the "effectively infinite" approach by providing a very large (8MB) physical buffer [72]. Infinite buffering offers the lowest overhead provided the real costs of buffering can be kept low. The next section talks about the performance cost of providing support for buffering in hardware.

Undeliverability is described here in terms of protection for correctness but there is also a related performance issue. If the message system does not demand prenegotiation of all resources, then undeliverability in terms of correctness becomes a receiver-side protection issue. However, there is also a similar performance issue: it might be beneficial to remove messages from the network (or, symmetrically, not to inject them) just to improve traffic flow within the network. Mukherjee, *et al* [58], found it beneficial to buffer messages at the receiver automatically in some applications.

Direct vs. Buffered Interfaces. Message passing network interfaces developed for high-performance parallel machines have taken two general approaches: direct and memory-based. Direct interfaces allow the processor to handle messages directly out of the network. Memory-based interfaces provide special hardware to extract messages out of the network and buffer them in memory; the processor then accesses the message buffers in memory. Although a definitive conclusion awaits further research, past research indicates that direct interfaces tend to be more efficient than memory-based interfaces. Direct interfaces that can be accessed at cache speeds offer even better performance [28]. For example, the CNI paper [56] showed that a direct, cache-level interface exhibited 50% higher bandwidth than their best interface placed on the memory bus. Direct interfaces are challenging to protect without sacrificing efficiency or seriously impairing the multiprogramming model. Therefore, one appeal of memory-based interfaces is that they may be protected through standard memory mapping mechanisms.

Figure 8-1 gives schematic views of the different approaches to message delivery. Figure 8-1a shows a direct interface with no buffering and Figure 8-1b shows a memory-based interface. Hybrid schemes are also possible. The two-case delivery system to be described in this thesis uses hardware for direct delivery and software for buffering as in Figure 8-1c. A two-case delivery system using all- or mostly-hardware by having the network interface manage memory itself is shown in Figure 8-1d. For instance, Mukherjee, *et al*'s $\mathrm{CNI}_{16}\mathrm{Q}_m$ [56, 57] interface provides both a fast path and a (potentially virtual) buffered path by using the network interface to buffer messages. Hybrid solutions will be discussed in more detail in Chapter 8.

Direct network interfaces, Figure 8-1a have been used in research machines [16, 7, 61, 1, 56] and one commercial machine, the CM-5 [45]. These interfaces feature low latency by allowing the processor direct access to the network queue. Direct NIs can be inefficient unless placed close to the processor. Anticipating continued system integration, we place our NI on the processor-cache bus. The CNI paper showed how to partly compensate for a more distant NI by exploiting standard cache-coherence support [56].

Memory-based interfaces, Figure 8-1b in multicomputers [6, 9, 66, 69, 72] and workstations [17, 19, 76, 77] provide easy protection for multiprogramming if the NI also demultiplexes messages into per-process buffers. Automatic hardware buffering also deals well with sinking bursts of messages and provides the lowest overhead (by avoiding the processors) when messages are not handled immediately.

Memory-based application interfaces provide low overhead when access to the network hardware is relatively expensive (true for most current systems), when latency is not an issue so that messages can be handled in batches, or when buffering is actually required. Increased integration of computer systems and the mainstreaming of parallel processing challenges these assumptions. On-chip network interfaces can have low overhead. Further, parallel programs frequently require low latency; so much so that they may require coordinated scheduling to keep latencies low and predictable. Coordinated scheduling has the effect of reducing the need for buffering. Buffering may still be required occasionally for protection reasons but the occasions are rare.

Direct interfaces tend to provide the best performance while buffering provides a means of solving the undeliverability problem. Table 2-1 summarizes the tradeoff between direct and memory-based interfaces by listing the operations required to receive messages using each of the control transfer mechanisms described in Section 2.1. For instance, messages received via interrupt using a direct interface (Figure 2-1, middle left) suffer the overhead of a user interrupt and the (minimal) overhead of reading the message payload from the tightly-coupled network interface. The direct interface is assumed to be accessed via load/store instructions that proceed at the speed of the level 1 cache. In contrast, the buffered interface suffers two extra overheads for reception via interrupt (Figure 2-1, middle right): the cost of reading data from memory and the cost of buffer management.

There are three other observations to make about this table. First, interrupts are demanding: it

Delivery of control	Delivery of data	
	Direct	**H/W Buffered**
Polling:	<Inter-poll time> Poll on network Read data from network Run handler	<Data placed in memory> <Inter-poll time> Poll on memory Read data from memory Run handler Buffer management
Interrupt w/upcall:	Interrupt Read data from network Run handler	<Data placed in memory> Interrupt Read data from memory Run handler Buffer management
Interrupt w/thread:		<Data placed in memory> OS: Interrupt OS: Schedule thread OS: Run handler thread Read data from memory Run handler Buffer management OS: Reschedule thread

Table 2-1. The sequence of operations used to receive a message for direct and memory-based data access modes using polling, upcall-based user interrupts and a thread signaled by an interrupt. Buffered operations differs from direct in the addition of overhead to store and fetch message data from memory and of buffer management overhead. Polling differs from interrupts by avoiding interrupt overhead, but adds the overhead of failed polls and the latency due inter-poll time. Thread-based interrupt handling differs from upcalls in the addition of scheduling overhead.

is easy for the overhead and latency of interrupts to become high, especially if a full thread model (Figure 2-1, lower right) is used. Second, polling can be either faster or slower than interrupts. The overhead of a successful poll is almost certainly lower than the overhead of an interrupt, but not all polls are successful and a polling application suffers additional latency due to the time between poll points. Third, most of the costs of control transfer and of memory access can be made to disappear if the application can be written so that messages are handled in *batches*. A memory-based interface then amortizes memory access costs over cache lines or could even apply prefetching. However, depending on a batching effect in a design reduces the usability of an interface. In other words, it is easy to provide communication for embarrassingly parallel applications using any technique; the challenge is to support the widest range of applications.

Between the demands of programmability and the consequences of protection, the best interface then depends on the situation:

Low latency is crucial	Buffering required because messages are undeliverable	Many opportunities for latency tolerance and batching
$\Downarrow$	$\Downarrow$	$\Downarrow$
Direct	**Memory-based**	**Either**

Summarizing this section, the ideal VNI architecture and implementation in terms of performance would suffer the minimum amount of extra cost in all situations over what is absolutely required to support the communication model and protection. Since direct and buffered interfaces have different strengths, the ultimate solution is likely to be a hybrid. The solution described in this thesis is one such hybrid. Other hybrid approaches are discussed in the related work presented in Chapter 8.

2.4 Problem Statement

The problem is to build a virtual network interface that addresses the three issues described in this chapter:

1. Programmability: The programming model must be an efficient target for the application programmer, a library developer or the compiler.
2. Protection: The system must coexist with flexible multiprogramming and the usual features of a multiuser machine (virtual memory) in order to support mixed workloads.
3. Performance: The implementation must be fast despite multiuser effects. Ideally the implementation is as fast as an implementation that is not burdened with multiuser features.

The design space is large and there is interesting work to be done in approaches that compromise one or more of the three points above. The goal of this work (and others), however, is to preserve all three points. The next chapter describes our solution.

Chapter 3

A Direct Virtual Network Interface

A virtual network interface must address the issues of programmability, protection and performance. This chapter introduces the heart of the thesis: a "direct" virtual network interface that achieves the three virtual network interface goals for message-passing communication with short messages. This chapter defines the programming and protection models and then overviews the architecture of the direct VNI that gives performance given the models. In other words, for the purposes of the thesis, the programmability and protection features are taken as design choices while the focus is providing good performance given these choices.

The direct VNI takes a novel approach to the VNI problem: the direct VNI provides a very low-level message model that optimistically maps to direct, user-level access to physical network hardware and then virtualizes that access through transparent emulation only when required for protection. The result is a solution that achieves performance near that of a single-user, all-hardware system with the flexible protection of a much more elaborate, software-based system.

The direct VNI achieves the goals of programmability, protection and performance:

- The direct VNI achieves the goal of programmability through its low-level model. The low-level model used is User Direct Messages (UDM), described here as well as by Kubiatowicz [40]. The particular model is arguably desirable, as we will show, but amounts to a design choice.
- The direct VNI provides protection that is compatible with multiprogramming, virtual memory and arbitrary scheduling policies, with some caveats. Isolation is based on labeling all the processes in an application with a single Group Identifier (GID) and permitting communication only between processes with the same GID. This protection mechanism is simplistic, but, combined with virtual memory and scheduling, produces the full set of undeliverability problems discussed in Section 2.2. The direct VNI architecture is thus applicable to systems with more general protection mechanisms.
- The direct VNI architecture achieves good performance because most messages proceed at hardware speeds, allowing performance approaching that of a dedicated, single-user machine without protection. The architecture achieves this goal via two techniques, two-case delivery and virtual buffering.

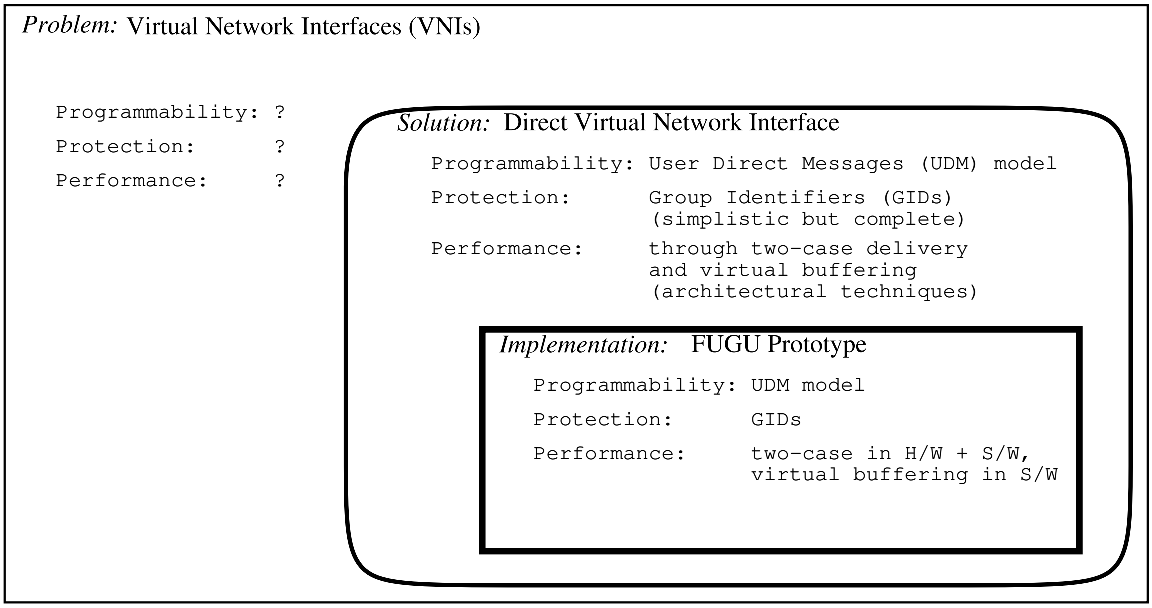


Figure 3-1. Summary of the problem, the solution to the problem at the architecture level, and the implementation of the architecture in FUGU.

Figure 3-1 depicts the relationship between the problem described in the last chapter, the solution architecture overviewed in this chapter and the architecture and implementation details to be described in subsequent chapters. Virtual Network Interfaces are network interfaces for multiuser machines. Any multiuser multiprocessor needs to address the VNI problem. The direct virtual network interface is the solution explored in this thesis. The focus of the thesis is on providing good performance given an aggressive, low-level communication model, represented by UDM, and a full set of protection features. The implementation applies the complementary architectural techniques of two-case delivery and virtual buffering in the FUGU prototype.

The remainder of this chapter is organized as a description of how the direct VNI addresses the issues of programmability, protection and performance as introduced in the previous chapter. Section 3.1 describes the UDM model which gives the direct VNI programmability. Section 3.2 describes the model of protection. The protection model used for this thesis is simplistic, but gives rise to the full set of protection-induced complications described in the previous chapter. Section 3.3 overviews the architecture of the direct VNI solution including the two-case delivery and virtual buffering techniques and describes how the solution achieves good performance. Section 3.4 concludes with a discussion of alternate design choices that arise at this point.

3.1 Programmability

The direct VNI addresses programmability by providing a low-level model, User Direct Messages or UDM. UDM is an abstract model for message-passing communication intended to represent a minimal but complete representation of the capabilities of a raw hardware interface. The UDM model is "programmable" in two senses. First, it is programmable in the sense that it is a natural, low-level target for a programmer, for a compiler or as a building block for other protocols (*e.g.,* send/receive, RPC) in a library. Second, although it could be implemented in terms of other primitives, UDM is sufficiently low-level to be implemented efficiently as a direct interface in hardware. UDM primitives correspond one-to-one with network hardware primitives and UDM is thus "programmable" in that it exposes the fundamental costs of the hardware.

The novelty of UDM with respect to other low-level models (such as Active Messages [78]) is that UDM defines the *control* transfer mechanisms along with the data transfer mechanisms as part of the model. UDM includes both polling and a user-level interrupt for message delivery notification. The key feature is a set of *atomicity* (interrupt disable) operations that allow an application using UDM to construct message receive code with the same flexibility and efficiency as an in-kernel device driver.

UDM provides all of the facilities commonly desired from fast message interfaces within a multiprocessor: low-overhead message construction and launch, as well as low-overhead reception via interrupts or polling. UDM allows the programmer to view the network hardware as a dedicated, user-level resource with effectively unlimited buffering. The buffering serves as an aide to deadlock avoidance in user protocols. The network is virtualized in the same sense that the CPU and memory are virtualized in a virtual machine. It is up to the hardware and runtime system to maintain these illusion.

As mentioned above, UDM has two major components, corresponding to data and control. First, the UDM model has a notion of *messages*, which are the unit of communication, along with operations to *inject* messages into the network at the source and *extract* them from the network at the destination. Second, and uniquely, UDM provides an explicit *atomicity* mechanism, which is a low-overhead, virtualized interrupt disable. The atomicity mechanism grants user code explicit control over the arrival of message interrupts, allowing a smooth integration of both polling and interrupts as mechanisms for notification of message arrival. The data and control parts of the model are discussed below, followed by a discussion of unlimited buffering used for deadlock avoidance.

Data Transfer Model. Data are transferred as the payload of a *message*. A message is a variable-length sequence of words. Two of these words are specialized: the first is an implementation-dependent routing header which specifies the destination of the message. The second specifies a handler to be run at the destination to receive the message, as in Active Messages. Remaining words represent the data payload and are unconstrained.[1]

The semantics of messaging are *asynchronous* and *unacknowledged*. At the source, messages are injected into the network at any rate up to and including the rate at which the network will accept them. The injection operation is *atomic* in that messages are committed to the network in their entirety; no "partial packets" are ever seen by the communication substrate [39]. This atomicity property is useful for multiprogramming because it allows the output interface to be multiplexed preemptively easily. Message injection can be viewed in the following fashion:

[1]The details of the data message constraints and layout are outside the scope of the UDM model. For concreteness, the following details are actually implemented in FUGU's direct VNI. First, the "destination" is a integer between 0 to $P-1$ corresponding to one of the P virtual processors in the current parallel application. Second, since the protection model (described in Section 3.2) limits messages to within a single application, the message "handler" is specified as the raw virtual address of the handler code. Third, the FUGU hardware supports messages of up to 16 words without the use of DMA.

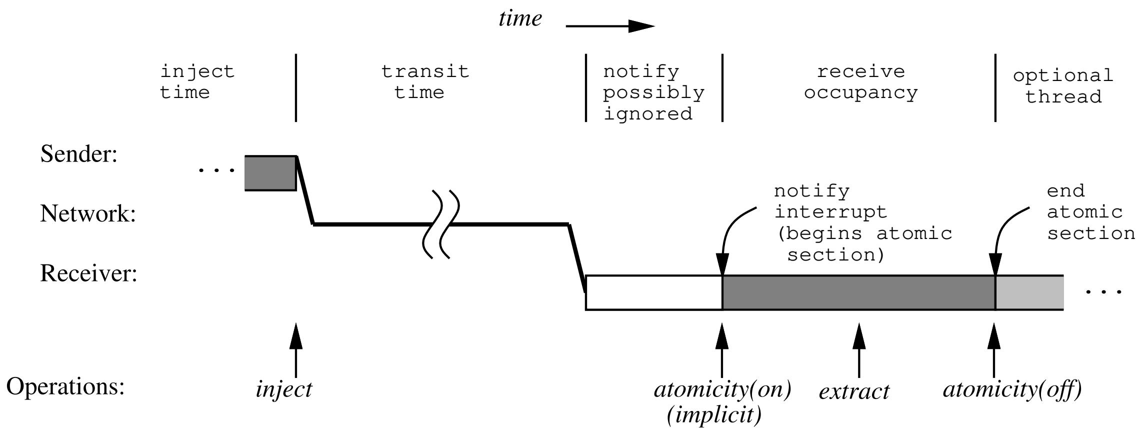


Figure 3-2. Message timeline for interrupt-based delivery on the fast path. The action at the sending node is at the top, at the network in the middle and at the receiving node at the bottom. The message is launched by an `inject` operation from a thread at the sender. After traversing the network, and after a possible delay due to interrupts disabled at the receiver, the arrival of the message invokes an interrupt handler at the receiver. The interrupt handler executes with interrupts disabled by default.

`inject` (*header, handler, word0, word1, . . .*)

The `inject` operation may block temporarily if the message system is unable to proceed for any reason (*e.g.,* resource contention in the network), but will eventually succeed. For performance reasons, blocking can be avoided by using a conditional, non-blocking version of `inject`, called `injectc`. Once a message has been injected into the network, the UDM model guarantees that it will eventually be delivered to the destination specified in its routing header.

At a destination, messages are presented sequentially for extraction.[2] A message is extracted from the network with an atomic operation that reads the contents of the message and frees it from the network:

`extract()` $\Rightarrow$ *(header, handler, word0, word1, . . .*)

Implicit in this syntax is that the message contents are placed directly in user variables without a redundant copy operation. The network provides a message available flag which can be examined to see if an `extract` operation will succeed. It is an error to attempt an `extract` operation when no message is available. Note that, in addition to the `extract` operation stated above, UDM provides a similar operation called `peek` which permits examination of the next message without dequeuing it.

By wrapping user-level network operations in `inject` and `extract` abstractions, UDM virtualizes these operations, permitting the underlying system to switch transparently between physical and virtual network access as needed. This is one of the central features of UDM, which we exploit for two-case delivery in later sections.

[2]Message ordering is implementation-dependent. FUGU preserves FIFO ordering between pairs of virtual processors.

Control Transfer Model. Control transfer is the other half of the UDM model. By providing full and explicit control over the reception of messages, UDM allows the programmer to interact with the network with the same flexibility and efficiency as an in-kernel device driver.

UDM assumes an execution model in which one or more *threads* run on each processor. Messages interact with those threads through the control transfer model. At the sender, the application transfers control of the message to the network in a single, atomic transaction. The `inject` may block for some period of time, but time amount of time is predictably small. At the receiver, message arrival time may or may not be predictable. UDM provides a polling operation, generally intended for situations in which message arrival is predictable, and a user-level message-available interrupt intended for situations when arrivals are unpredictable.

Polling and interrupts are stitched together with an *atomicity* operation, which disables and enables the user interrupt.

`atomicity(`*on* | *off*`)`

Setting atomicity *on* disables the user interrupt. Periods of execution in which interrupts are disabled are called *atomic sections* because they execute atomically with respect to the message-available interrupt. When atomicity is on, notification is entirely through the message available operation:

`message_available()` ⇒ *true* | *false*

In this mode, the currently running thread must poll the network interface with `message_available` and extract messages with `extract` as they arrive.

In contrast, when atomicity is *off*, the existence of an input message causes a user interrupt. The user interrupt causes the current thread to be suspended and an independent *handler*, to be initiated. The sequence of events is illustrated in Figure 3-2. The handler begins execution in an atomic section (*i.e.,* with interrupts disabled), at the handler address specified in the message. A handler is required to extract at least one message from the network before exiting or re-enabling interrupts.

A handler is not a full thread. However, when a handler exits, some runnable thread is resumed. This thread might be a thread awakened by the handler, a thread created by the handler, or the interrupted thread; the exact scheduling policy is defined by a user-level thread scheduler, not by the UDM model.[3] In particular, UDM is compatible with extremely lightweight thread systems in which message handlers are occasionally or routinely converted to threads after executing only the minimal code required to communicate with the network interface.

The key to control transfer in UDM is the `atomicity` operation. User-level atomic sections permit user code to construct interrupt handlers, to poll, and to construct critical sections that are atomic with respect to interrupts. This level of control over interrupts is typical, if ad hoc, in kernel-level device drivers. Providing this control at user level allows application code to interact with the network interface with the same efficiency and flexibility as kernel code. As with `inject` and `extract`, the `atomicity` operation is an *abstraction* for a physical interrupt disable: in

[3]The FUGU thread scheduler supports all three options.

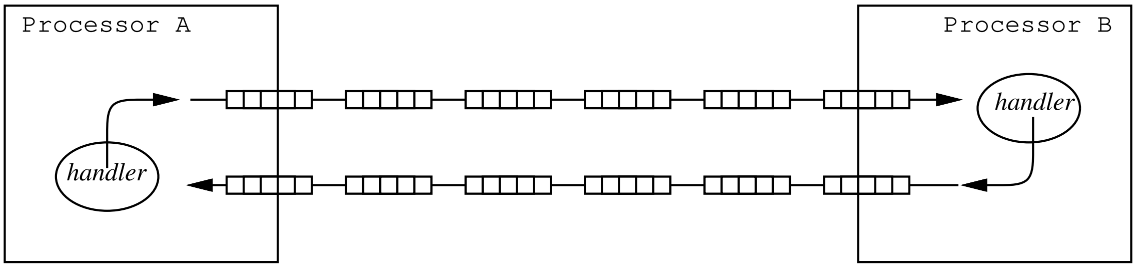


Figure 3-3. Code that sends a message in a handler may create a circular dependence and thus deadlock. Here, both processors A and B are waiting to `inject` messages, but the network is already filled with messages. Some message must be received before either `inject` operation can proceed.

the common case, the user's requests for atomicity interact directly with the network hardware to defer interrupts. When necessary, however, these requests may be virtualized to free up the physical interface while maintaining the illusion of atomicity to the user.

Deadlock Avoidance. As mentioned earlier, the UDM model provides a single network with conceptually "unlimited" buffering. Unlimited buffering is provided as a means to help avoid deadlock in user-level protocols.

All communication protocols must address the possibility of deadlock. For instance, when implementing a remote memory read protocol atop UDM, it is convenient to send a "reply" message from within the atomic section of a "read request" handler. Figure 3-3 illustrates a deadlock situation with two processors waiting to send such a reply message while each is in an atomic section. The processors are deadlocked because the network between the two processors happens to be completely full of (unrelated) messages in both directions. A circular dependence has been created involving the two handlers and the two paths through the network.

Deadlocks situations can be avoided by writing programs that avoid deadlock in ad-hoc ways. However, it is useful to have the communication system provide some help. The infinite buffering approach in UDM is one form of help. In the direct VNI, a deadlock is eventually *detected* by the timeout on network blockage. Timeouts cause messages to be routed through the buffered path. Buffering breaks the circular dependence that caused the deadlock. The approach of detecting deadlock when it happens is based on the assumption that deadlock situations are rare. The Alewife machine provides software buffering in "Network Overflow" to break deadlocks in it's hardware shared memory system [40].[4]

It is insightful to compare the deadlock avoidance strategy provided by the direct VNI to other possibilities. Many systems (e.g. Active Messages [78]) define separate logical "request" and "reply" networks and handlers to solve exactly the situation depicted in Figure 3-3. The two logical networks are can be two actual, physical networks or just one network with two priorities where reply messages use the higher priority. The two logical networks are used with the following discipline: request handlers are permitted to send reply messages but reply handlers are not permitted to send any messages. Given this discipline, request handlers execute atomically with respect to request

[4]Network overflow is a limited form of two-case delivery; its usefulness inspired the more general usage in the direct VNI.

handlers (but not with respect to reply handlers). Reply handlers execute atomically with respect to reply handlers. With separate request/reply networks, a communication protocol is deadlock-free as long as it adheres to the request/reply discipline.

However, not all protocols are easily mapped to the request/reply discipline. For instance, the "remote-writer" optimization in a shared-memory protocol results in a three-way trip. Solving deadlock in such a protocol requires three logical networks. The Remote Queues [8] model defines communication through multiple named queues. An application can define as many queues (which correspond to multiple logical networks) as necessary to avoid deadlock, presuming the queues are big enough.

UDM defines a programming model with efficient data transfer and control operations and with the convenient fiction of unlimited buffering for deadlock avoidance. The model is described in terms of virtualized networks with virtualize resources. Multiplexing the virtual networks and virtual resources onto physical networks and resources requires protection, described next.

3.2 Protection

The protection issue in a virtual network interface arises from the several features that are standard in a multiuser machine: multiprogramming, demand paged virtual memory and priority-based scheduling. The main consequence of protection, as described in Section 2.2 is the undeliverability problem: the destination process may not always be able to receive a message at a given point in time, yet, for protection, the system must be able to empty the network to allow other unrelated messages to proceed.

The direct VNI used in this thesis incorporates a simple model of protection and the FUGU system includes limited virtual memory and a priority-based scheduler. However, the simple protection model and other features give rise to the full suite of undeliverability situations, so the direct VNI solutions are fully general.

The protection model associated with UDM is simply that each application views a private, isolated virtual network. Isolation is achieved by tagging all messages with a "Group Identifier" (GID) corresponding to the group of messages that make up a parallel application. An application's messages are indelibly tagged with the GID at `inject` time and the application may only see incoming messages that have a matching tag. Figure 3-4 shows a typical layout of applications with this minimal protection model. A parallel application consists of a set of processes with the same GID spread across multiple processors. Processes with the same GID on different processors are permitted to communicate with UDM. Communication between processes on different processors with differing GIDs is not supported.[5]

Beyond GIDs, there is no other protection of messages in the direct VNI. In particular, messages are addressed with unchecked physical processor numbers, so it is possible to misaddress messages. Such misdirected messages are detected by the operating system at the receiving processor. While this receiver-based protection is sufficient, a real system would likely support destination address

[5] Different processes on the *same* processor may communicate via conventional uni-processor inter-process communication (IPC) mechanisms. The GID-based protection model supports client-server computing with the restriction that the client and/or the server must be distributed so that cross-domain communication is local to a processor.

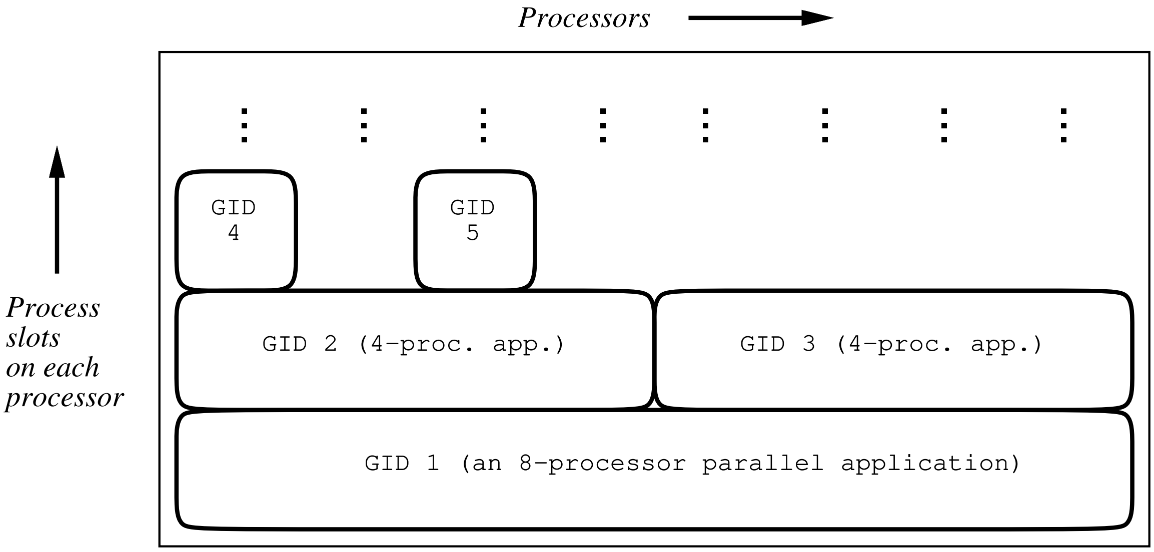


Figure 3-4. Protection between applications is based on Group Identifiers (GIDs). All the processes in an application are labeled with a single GID. A process may only send messages to another process with the same GID.

translation, both to eliminate misdirected messages and to ease process migration.

The direct VNI interacts with the virtual memory system and the scheduler system in the FUGU system. Again, these systems are limited in capability, but we use them without loss of generality because they give rise to the same undeliverability situations as more complex systems. The virtual memory system generates page faults which may delay message reception for an extended period. The scheduler is capable of performing priority-based scheduling independently on each processor, a technique that can cause many mismatched messages.

GID-based protection is minimalist and a commercial system would almost certainly provide more flexible and expressive mechanisms. However, the GID-based protection, along with virtual memory and the scheduler, produces the full range of undeliverability situations for the direct VNI. The next section describes how the direct VNI architecture provides performance given the model and the effects protection.

3.3 Performance

The direct VNI supports programmability via the UDM model and protection as described above. The goal of the direct VNI architecture is to provide good performance given these design choices. The architecture makes use of two-case delivery and virtual buffering techniques to achieve that goal. Each technique has performance benefits but also a potential performance tradeoff. This section briefly explains the ideas and explains the sources of the tradeoffs. We conclude by summarizing the implications of the architecture to the programmer in an informal model.

Delivery of control	Delivery of data		
	Direct	**H/W Buffered**	**S/W Buffered**
Polling:	<Inter-poll time> Poll on network Read data from network Run handler	<Data placed in memory> <Inter-poll time> Poll on memory Read data from memory Run handler Buffer management	OS: Interrupt OS: DMA data to memory <Inter-poll time> Poll on memory Read data from memory Run handler OS: Buffer management
Interrupt w/upcall:	Interrupt Read data from network Run handler	<Data placed in memory> Interrupt Read data from memory Run handler Buffer management	*(not supported)*
Interrupt w/thread:	*(not supported)*	<Data placed in memory> OS: Interrupt OS: Schedule thread OS: Run handler thread Read data from memory Run handler Buffer management OS: Reschedule thread	OS: Interrupt OS: DMA data to memory OS: Schedule thread OS: Run handler thread Read data from memory Run handler OS: Buffer management OS: Reschedule thread

Table 3-1. Delivery operations for direct, hardware-buffered and software-buffered interfaces with three control transfer techniques. Two-case delivery makes use of a combination of the direct and software-buffered techniques. The hardware-buffered technique is included for comparison

3.3.1 Two-Case Delivery

The direct VNI architecture reconciles communication performance with the undeliverability problem from Section 2.2 by using two-case delivery. Two-case delivery provides both a direct path and a buffered path for message reception, giving the message system the advantages of both approaches. At the source of a message, the `inject` operation always makes use of a user-accessible, direct interface in hardware. At the destination of a message, the `extract` operation ordinarily uses the direct interface but may sometimes be required to receive a message via a buffer in memory. The direct case gives the interface speed while the buffered case gives the system robustness.

Two-case delivery addresses performance given the programming and protection models in two ways:

1. The direct case gives the message system high speed. When messages are commonly received via the direct path, the message system exhibits performance that approaches that of an entirely unprotected, single-user machine.
2. Splitting delivery into two cases allows much of the implementation complexity to be handled in software. The hardware support is kept small and the hardware design is focussed on the performance of the common case.

A performance tradeoff arises in two-case delivery because, since the direct access mode gives the interface speed, we take the opportunity to save on hardware complexity and implement the buffering mode largely in software. Two-case delivery thus performs better than a competitor that always buffers messages in hardware only if the direct case is the common case. Table 3-1 extends Table 2-1 from the previous chapter with the sequences of operations used to receive messages through software buffering. The operations for software buffering are similar to those for hardware buffering with the addition of an operating system interrupt handler used to insert new messages into the buffer.

Polling delivery in two-case delivery uses either the direct case with polling (upper left in Table 3-1) or software-buffered case with polling (upper right). Interrupt delivery uses either the direct case with upcall-based interrupts (middle left) or the software-buffered case with thread-based interrupts (lower right). The key to two-case delivery, as will be described in Chapter 4, is that the system provides the same UDM model transparently in both direct and software-buffered modes. The programmer writes one program and the operating system decides how to manage the network interface modes.

Comparing the direct and software buffered cases in Table 3-1 to the hardware buffered case makes clear the tradeoff between two-case delivery and a system that always buffers messages in memory using hardware. Direct messages cost less than hardware-buffered messages, but software-buffering costs yet more. The premise of two-case delivery is, then, that there exist situations in which buffering is *required* to solve the problem of undeliverable messages, but that those situations are (or can be made to be) uncommon. We will show that the premise is true in Chapter 7.

Buffer Scheme	Application Consequences	System Consequences
No buffering	Must deal with arbitrary dropped messages.	Trivial.
Small, fixed	Use protocol or proof to avoid exceeding the buffer limit.	Any runnable application must have a small physical buffer.
Large, fixed	Buffer consumption is not a practical problem (if the buffer is large enough).	Any runnable application must have a large physical buffer.
User-pinned	Like small or large but the application can pick the size.	Physical buffers must be supplied but many applications may need only small buffers.
Virtual	Like large; buffer size is a performance issue, not a correctness issue.	The system may allocate and deallocate buffer space on demand but must avoid deadlock.

Table 3-2. Options for implementing the buffered part of two-case delivery ranging from no buffering to virtual buffering. Virtual buffering gives the programmability advantage of a very large buffer without the physical memory requirement.

3.3.2 Virtual Buffering

Two-case delivery is complemented by the second technique, virtual buffering. Virtual buffering stores the buffered messages in virtual memory allocated on demand. Buffering in virtual memory gives the application and the system several important and unique advantages over buffering in physical buffers. A cost is that the virtual buffer performance will generally degrade as more buffering is used. Although applications will rarely need to limit buffer consumption for correctness, some may still need to limit buffering for performance.

Virtual buffering addresses performance given the programming and protection models in three ways:

1. It stores messages in paged virtual memory, which makes it practical to provide the "unlimited buffering" anti-deadlock feature of the UDM model.
2. It guarantees message delivery, which allows the message system to avoid all protocol overhead in the fast case. In contrast, a system with small, fixed buffers must always account for the limited buffer space, which adds complexity and overhead.
3. It allocates and may page buffer storage on demand, which allows the operating system to manage and presumably to minimize physical memory requirements for buffering.

The benefits and costs of virtual buffering are made plain by comparing virtual buffering to alternatives. Table 3-2 lists options for buffering used with two-case delivery, including the degenerate case of no buffering. If the amount of buffering is fixed and "small", the application must explicitly deal with that restriction. The easiest way is for the application to use a low-level protocol that limits the buffering required. The programmer might also prove that the program can never require more than a fixed amount of buffer space. If the amount of buffering space is fixed but "large", the programmer can just trust the program not to exceed the buffer space. Larger buffers are easy on the programmer but hard on the operating system, which must now guarantee space to every application that might communicate. User-directed pinning of pages can ease the amount of physical buffer space required as long as most programs do not pin many pages.

Compared to physical buffering alternatives, virtual buffering addresses both the application programmer's problems and the system's memory consumption problem at the cost of some extra complexity in the operating system. With virtual buffering, the amount of available space can be considered unlimited. Buffer consumption becomes a matter of performance rather than of correctness for an application. From the operating system's perspective, pages used for buffering are dynamically allocated and pageable, freeing up physical memory. Compared to a system with a large (effectively infinite) physical buffer, a performance cost of virtual buffering is that virtual memory management costs will tend to degrade performance as more buffer space is used. The premise of virtual buffering, like two-case delivery, is that it is important to be able to buffer in some situations, but that those situations are uncommon.

3.3.3 Programmer-Visible Performance Tradeoffs

The UDM model and the direct VNI architecture give the programmer a lot of freedom. The UDM model gives each application the illusion of a private network, virtualized resources and control over every logical message passed through the network. In practice, there are two subtle performance issues that the programmer faces when using the direct VNI.

- The combination of one-way, asynchronous messages and unlimited buffering means that the problem of flow control has largely been pushed up to the level of the UDM user. This design decision is a benefit – the application does not incur the cost of unnecessary flow control protocol overhead – but also a burden. An application must either limit the flow of messages by construction or make use of a user-level library that includes a flow control protocol. The message system includes a coarse form of flow control ("overflow control", described in Chapter 5) but only as a last resort defense against temporarily or permanently runaway applications.
- The buffering system breaks deadlock scenarios, but use of buffering incurs performance costs. For instance, an application is free to attempt to send a message from within an atomic section, but if a deadlock occurs it is detected by a timeout only after some amount of delay. Further, an application may choose to depend on the existence of a fixed amount of buffering, but buffering has a real cost. An astute UDM programmer will depend on the buffering system only in the "optimistic" sense in which is was intended, *i.e.,* only when the chance of an actual deadlock is known to be low.

Effectively, correctness is separated from performance. It is possible to write an application with little attention to the effects of the message system and then concentrate effort on the performance of sections that matter. There are two performance effects to keep in mind. First, the buffered delivery path incurs extra cost. Second, the virtualization of buffering can introduce even higher costs when

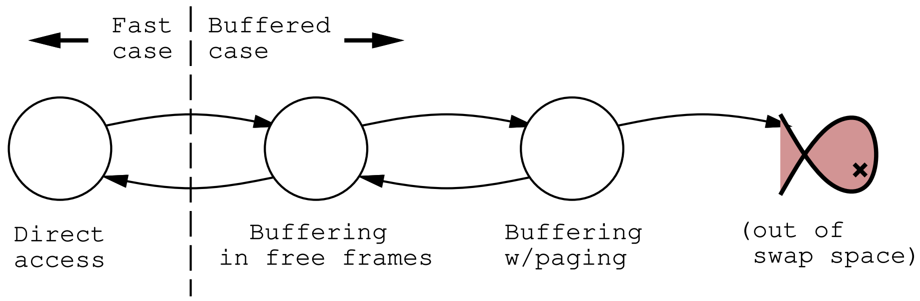


Figure 3-5. Direct virtual network interface operating modes as a state diagram. In the fast, common case, at left, message reception is handled entirely in software. Buffering has additional costs which increase as demands for buffer storage space increase.

physical memory resources are low. These effects are described below and quantified in Chapters 4 and 7.

Figure 3-5 summarizes the operating modes of the FUGU system as a state diagram model. In the ordinary case, on the left, UDM messages are extracted directly out of the hardware NI at full speed using polling or upcall-based interrupts. Next, buffering, when invoked, adds some cost but operates similarly to a memory-based system as long as buffer space is available in physical memory. Third, virtual buffering system can continue to buffer messages for an application even when physical buffering space is exhausted by paging and buffering simultaneously. This case is quite slow but breaks potential deadlocks arising from dynamic allocation of buffer space. Finally, at the right, it is possible to exhaust the application's virtual buffer, but only by exhausting the application's swap space.

The messages to the programmer are two. First, some applications will give better *performance* if a little protocol overhead is added for the sake of avoiding buffering. We have no direct examples of this effect; all our sample applications (in Section 7.4) avoid buffering naturally. Second, it is possible to write programs that consume even "effectively unlimited" buffers. Like any system that supports unacknowledged messages, the FUGU programmer (or library) must either keep the message generation rate below the message consumption rate or provide synchronization to do so in order to avoid filling up the buffer.[6]

3.4 Discussion

This chapter has described the programmability, protection and performance aspects of the direct virtual network interface architecture. The model and protection features are accepted as design choices while the focus is on providing performance in the face of those choices. The two-case delivery and virtual buffering techniques described above will be revisited in detail in the next chapter. At the level of the architecture, however, several alternate design decisions or extensions are possible.

[6]Further, given the in-order delivery used in the FUGU implementation of the direct VNI, any flow control scheme must make sure to *clear* the buffer before reenabling messages. Flow control schemes based on, for instance, fixed-sized windows may never clear the buffer.

Bulk Transfer. The direct VNI as described here is oriented to support for small messages where the primary consideration is low latency from processor to processor. Large messages for bulk transfer have the different goal of high bandwidth from memory to memory. The direct VNI is compatible with efficient solutions to bulk transfer (see [50]). Appendix A describes limited extensions for bulk transfer actually implemented and used in FUGU.

Source Buffering. The direct VNI applies buffering only at the receiver and largely for reasons of protection. It is possible to invoke buffering for performance reasons as well. Further, buffering for performance might use buffering at the sender as well as buffering at the receiver. Transparent buffering at the sender could be implemented with very similar techniques to the ones described here used at the receiver.

User Pinning. Table 3-2 lists user pinning as a means of managing physical resources. User pinning could be extended to be as flexible as virtual buffering. The key trick is that the application (or library) has to be able to renegotiate for buffer space at the moment a message arrives to avoid dropping any messages.[7] Alternatively, virtual buffering could (and should) be extended to take advantage of advice from sophisticated applications that know the amount of buffering needed. A dynamic default policy plus a means for using advice is probably the right way to approach any resource management problem.

Bulk transfer, source buffering and user pinning are possible extensions to the direct VNI but do not change its basic architecture based on two-case delivery and virtual buffering. The next two chapters describe the two-case delivery and virtual buffering architectural techniques in detail, respectively.

[7] FUGU's Exokernel operating system does in fact use user-level virtual memory in exactly this way.

Chapter 4

Two-Case Delivery Technique

The direct virtual network interface uses two-case delivery and virtual buffering techniques to achieve the goal of performance given the constraints of programmability and protection, as described in the previous chapter. This chapter describes the first of the two techniques: two-case delivery. We describe the technique in detail using examples from the FUGU implementation for concreteness. The next chapter describes virtual buffering.

Two-case delivery provides performance by optimistically giving user-level access to data and user-level atomicity control in hardware. The UDM model described in the previous chapter provides the abstract interface to the communication mechanism. The send-side of the UDM abstraction is implemented in hardware in the direct VNI. The receive-side is implemented two ways, in hardware and in software, corresponding to the two cases in two-case delivery. The runtime system then choses between the two cases according to system conditions.

Two-case delivery provides two alternate means of receiving a message: a direct path intended for minimum overhead and a buffered path included to solve the problem of undeliverability. The mechanics of the two paths are illustrated in the form of timelines in Figures 4-1 and 4-2. In the fast case, the network interface hardware is controlled directly by the application. The application is notified of message arrival by polling or by a user interrupt. The application reads the message from the network interface and implicitly blocks the network while doing so. In essence, the application reads data directly "from the wire" for lowest overhead. In the buffered case, an operating system handler intercepts incoming messages and places them in a queue in memory. The application then polls for messages in the queue or receives messages via "interrupts" orchestrated by an operating system thread.

The basic assumption is that messages ordinarily arrive under perfect conditions so that they may be delivered via the fast path. Some "undeliverable" messages may in fact be deliverable under some conditions. For instance, a message for another process may be delivered via a cross-domain upcall. If everything fails, the runtime system switches to a buffering mode with the same abstract interface but a completely different implementation.

This remainder of the chapter is organized around the parts of two-case delivery. First, Section 4.1 describes the fast case by giving an ISA-level description of the memory-mapped network interface hardware and its use. The fast case includes hardware protection to support multiprogramming. The central feature of the fast path is the revocable interrupt disable mechanism that permits protected

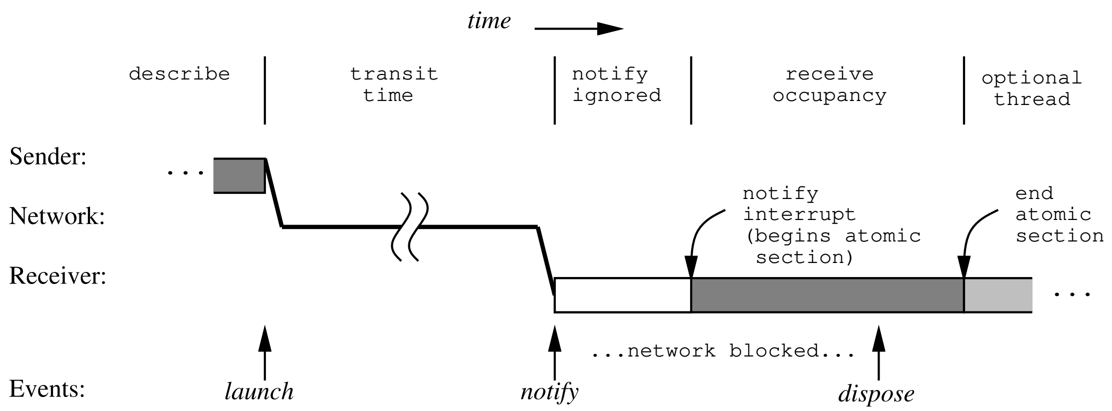


Figure 4-1. Message timeline for interrupt delivery on the fast path. The message is launched by a `launch` instruction which commits the body of the message to the network atomically. After traversing the network, the arrival of the message raises a *message-available* interrupt. The interrupt handler removes the message from the network atomically with a `dispose` instruction.

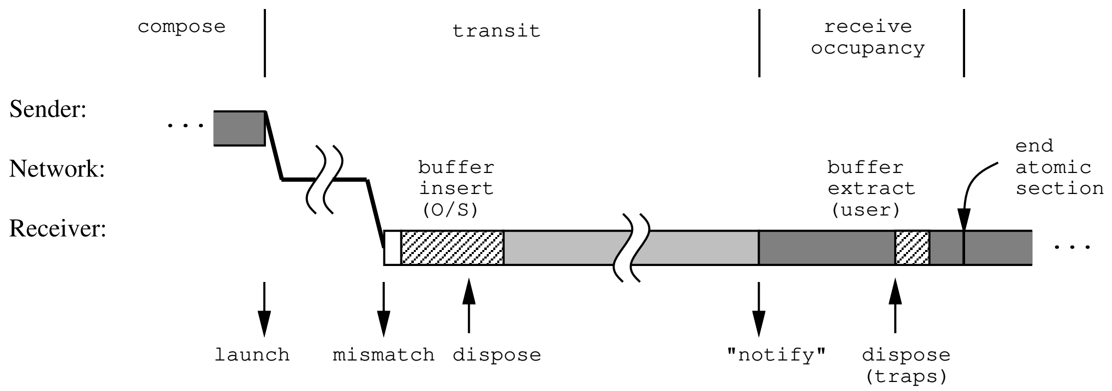


Figure 4-2. Message timeline for the buffered path.

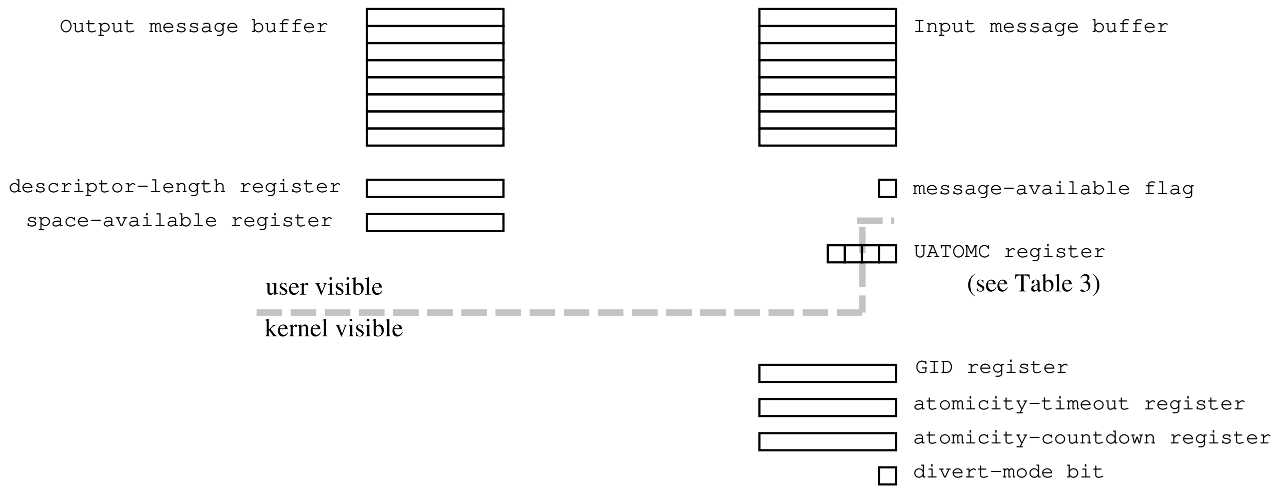


Figure 4-3. Direct virtual network interface registers.

control over atomicity for user interrupts and direct polling. Section 4.2 describes the buffered delivery case and shows how it provides semantics identical to the fast case. Section 4.3 puts the two cases together by describing how the interface to the two modes is kept transparent and how transition is invoked. Finally, Section 4.4 overviews the research questions raised by the two-case delivery technique and discusses possible alternate implementations.

4.1 Direct Access Path

The direct VNI consists of a set of memory mapped registers shown in Figure 4-3, a set of atomic operations listed in Table 4-1 and a set of interrupts and traps listed in Table 4-2. The operations are implemented as instructions in FUGU but might be encoded as writes to additional memory-mapped registers. The user-level registers, operations and the *message-available* interrupt are manipulated directly by user code when the fast mode is enabled, *i.e.,* under ordinary conditions. The kernel registers and the rest of the interrupts and traps both control the transition from fast to buffered mode in response to exceptional conditions and support operation in buffered mode. Further discussion of buffering is deferred to Section 4.2.

Send and Receive. The `inject` operation of the abstract model is decomposed into a two-phase process of *describe* and *launch*, as in [39]. To send a message, an application first writes all of the message data into the output message buffer starting at zero offset from the beginning of this buffer. The send buffer is special in that store operations at a given offset will block if the network is currently unable to accept a message as large as one that is implied by the offset. The *space-available* register, used to implement `injectc`, reflects the number of send buffer words that may be written without blocking. The buffer in our implementation is limited to 16 words; larger messages utilize an associated user-level DMA mechanism [50] (described in Appendix A).

Once the message has been completely described, it is guaranteed that the network will accept it. At that point, the message is injected into the network with an atomic `launch` instruction whose operand reflects the length of the message. The `inject` operation remains atomic because `launch`

Operation	Description
`launch(N)`	**If** *header* **==** kernel message **then** cause a *protection-violation* trap. **elseif** *descriptor-length* $>$ 0 **then** Commit an `N`-word message to the network; set *descriptor-length* **:=** 0

`dispose`	**If** *divert-mode* set **then**
	cause a *dispose-extend* trap,
	elseif *message-available* not set **then**
	cause a *bad-dispose* trap,
	else
	delete current incoming message.

`beginatom(MASK)`	set *UAC* **:=** (*UAC* $\vee$ `MASK`).

`endatom(MASK)`	**If** *dispose-pending* is set **then**
	cause a *dispose-failure* trap.
	elseif *atomicity-extend* is set **then**
	cause an *atomicity-extend* trap.
	else
	set *UAC* **:=** (*UAC* $\wedge$ ($\sim$`MASK`))

Table 4-1. Direct virtual network interface operations.

Interrupt/Trap	Event Signaled
message-available	User interrupt: raised when a message is available for reading
mismatch-available	Interrupt: message available with mismatched GID (or all messages when *divert-mode* is set)
atomicity-timeout	Interrupt: atomic section timer expired
atomicity-extend	Trap: optional at end of atomic section
dispose-extend	Trap: optionally triggered by `dispose`
dispose-failure	Trap: triggered by `dispose` when application fails to free message
bad-dispose	Trap: triggered by `dispose` with no pending message.
protection-violation	Trap: user access to kernel registers *or* user `launch` with kernel message

Table 4-2. Direct virtual network interface interrupts and traps.

is atomic: at any point before launch, the contents of the output buffer may be transparently unloaded and later reloaded if necessary for a context switch. The *descriptor-length* register reflects the number of words in the buffer that would need to be swapped at any given time. After a `launch`, data in the send buffer may be altered immediately without affecting any previously injected messages.

The `extract` operation is decomposed in an analogous way. The contents of the next pending message are made available beginning at offset zero from the input message buffer. Access to data within the message is performed by reading data from the buffer, then executing a `dispose` instruction. The `dispose` operation then exposes the next message, if available, for extraction. Atomicity of `extract` is maintained because `dispose` is atomic.

The application is notified of the arrival of a new message either by a *message-available* interrupt (converted to a user-level interrupt) or by explicitly polling the *message-available* flag in the network interface. The selection between the two modes is performed by the revocable interrupt disable mechanism described below.

Protection. The network interface hardware includes protection mechanisms sufficient to enable multiprogramming. The emphasis is on keeping the common case fast while reflecting all other cases to software. There are three hardware facilities used:

1. Isolation between users is maintained by labeling all messages with a Group Identifier (GID) stamped by hardware at the sender and checked by hardware at the receiver.
2. The duration of a user interrupt or upcall handler is bounded by a timeout timer (discussed below).
3. A reserved, second network exists for occasional use by the operating system in situations otherwise subject to deadlock (see Section 4.2).

The GID labels a group of processes (virtual processors) operating together, *e.g.,* the processes corresponding to the processors in a parallel application. UDM provides the simplest GID-based demultiplexing system in hardware: at the receiver, if the GID in the header matches the GID of the current application, the application is notified of message arrival via the *message-available* interrupt or via the *message-available* bit for polling. Otherwise, a *mismatch-available* interrupt is generated, allowing operating system software to perform the rest of the demultiplexing in this uncommon case.

As described in Section 3.2, the direct VNI applies all protection at the receiver. The sender is controlled only indirectly by the global scheduler. Messages directed to incorrect destinations are detected because they cause *mismatch-available* interrupts. The operating system handler then uses the global scheduler to find and to perform the appropriate action against the offending sending application.

Revocable Interrupt Disable. As was discussed in Section 3.1, UDM includes an explicit notion of *atomicity*, *i.e.,* the ability to disable message interrupts. The atomicity mechanism is an abstraction. Although the user is presented with the illusion of a dedicated network interface, there are several reasons not to allow the user to directly block the network interface by disabling interrupts:

- Malicious or poorly written code could block the network for long periods of time, preventing timely processing of messages destined for the operating system or for other users, *even on other nodes*.

User Controls	Description
interrupt-disable	When set, prevents *message-available* interrupts. In addition, if a message is pending, enables atomicity timer; `dispose` operation briefly disables (*i.e.,* presets) timer.
timer-force	When set, enables atomicity timer unconditionally.

Kernel Controls	Description
dispose-pending	Set by OS in the *message-available* stub, reset by `dispose`. See `endatom` in Table 4-1.
atomicity-extend	Requests an *atomicity-extend* trap. See `endatom` in Table 4-1.

Table 4-3. Detail of individual flags in the User Atomicity Control *(UAC)* register.

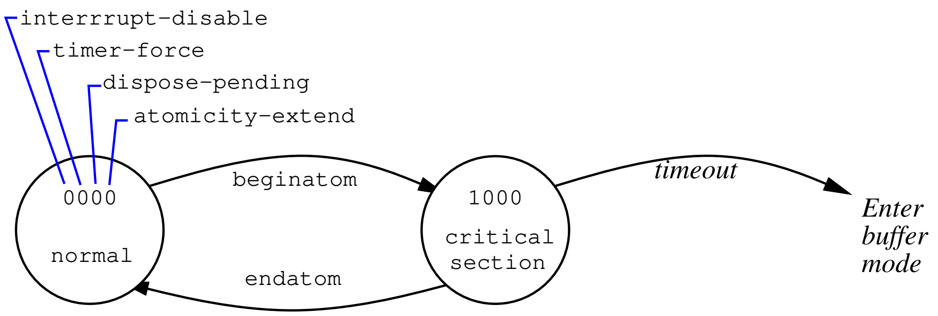
- When the user is polling, the system as a whole may still need to receive messages on the local node via interrupts to ensure forward progress.
- The operating system must demultiplex messages destined for different users. This process should be neither visible to nor impeded by any particular user.

At the same time, the user should enjoy similar efficiency in the common case to the operating system, extracting messages directly from the network interface.

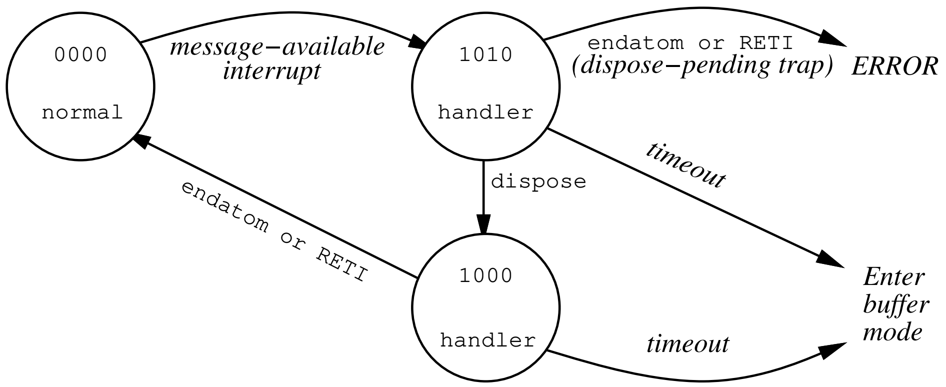
These problems are solved by implementing atomicity through a *revocable interrupt disable* mechanism. The main idea behind this mechanism is that the user is allowed to *temporarily* disable hardware message interrupts. As long as the network continues to make forward progress, the user is allowed to continue disabling interrupts. Should a message stay blocked at the input queue for too long, the system *revokes* the interrupt disable privileges, switching from physical atomicity (*i.e.,* disabling of the actual queue) to virtual atomicity (*i.e.,* buffering messages in memory and hiding them from the user until the atomic section is exited). Thus, the revocable interrupt disable mechanism can trigger an explicit entry into buffering mode.

The central feature of the revocable interrupt disable mechanism is a dedicated atomicity timer which can be used to detect lack of forward progress. By dedicating this timer the system can provide low-cost "instructions" which reset the timer and which enable the timer when the network is blocked. In addition, the atomicity mechanism is designed to affect messages destined for the currently scheduled user: when messages destined for other users (or the operating system) arrive at the head of the queue, they cause interrupts to the operating system, even if the user has requested atomicity. Further, when the buffered path is in use, all messages interrupt the operating system, regardless of whether the user has requested atomicity.

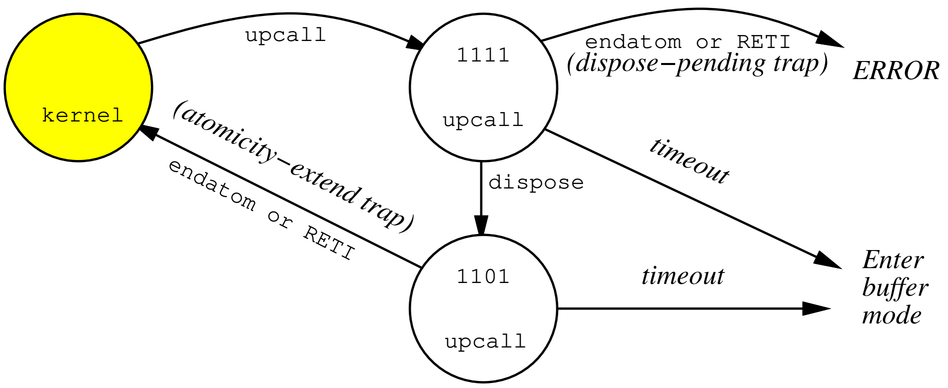
Control over user-level interrupts is implemented with four atomicity control bits in the User Atomicity Control *(UAC)* register which are manipulated via the `beginatom` and `endatom` operations. Table 4-3 details the individual flags in the *UAC* register. Two of the bits are modifiable only in kernel mode and are configured by the hardware or kernel code before giving control of the processor to the user. The other two bits can be set and reset by the user via `beginatom` and `endatom`, respectively. Under certain conditions, noted in Table 4-1 (but generally whenever either of the kernel bits is set), `endatom` executed in user mode will trap to return control to the operating



(A) Critical Section / Polling



(B) User−Level Interrupt



(C) Upcall with Priority Inversion

Figure 4-4. Three revocable interrupt disable examples. User-level nodes are labeled with the *UAC* state at the top. Kernel-level nodes are shaded.

Item	FUGU kernel mode (cycles)	FUGU hard atomicity (cycles)	FUGU soft atomicity (cycles)
Message Send			
Descriptor construction	6	6	6
`launch`	1	1	1
send total:	**7**	**7**	**7**
Message Receive (interrupt)			
Interrupt overhead	6	6	6
Register save	16	16	16
GID check	–	10	10
Timer setup	–	1	13
Virtual buffering overhead	–	8	8
Dispatch (+ upcall)	10	13	13
subtotal:	32	54	66
Null handler (w/`dispose`)	5	5	5
Upcall cleanup	–	10	10
Timer cleanup	–	1	17
Register restore	17	17	17
interrupt total:	**54**	**87**	**115**
Message Receive (polling)			
Poll	3	3	
Dispatch	5	5	
Null handler (w/`dispose`)	1	1	
polling total:	**9**	**9**	*n.a.*

Table 4-4. Cycle counts to send and receive a null message. Add 3 cycles per argument to the send cost and 2 cycles per argument to the receive handler cost for non-null messages. The "soft atomicity" numbers include overhead to emulate the atomicity mechanism on the first silicon CMMU and the current simulation system.

system.

The atomicity timer mechanism is comprised of a decrementing counter and a preset value, *atomicity-timeout*. While the timer is *disabled*, the counter is preset to the *atomicity-timeout* value. When the timer is *enabled*, the counter decrements for each *user* cycle, flagging an *atomicity-timeout* interrupt if it reaches zero. The counter is enabled during atomic sections by the user *UAC* bits, as described in Table 4-3.

The use of the revocable interrupt disable mechanism is best illustrated by example. Figure 4-4 illustrates several different uses of the atomicity mechanism: for polling, user-level message interrupts, and for user-level message interrupts during priority inversion. The timeout timer provides a bound on the user control over the processor and the network. Note that the exact timeout value is a free parameter that may be changed without affecting correctness. Paths through this figure which exit to the left represent fast-path usages of atomicity, while exits to the right represent entry into buffer mode (or errors).

Fast Path Performance. The fast path in the direct virtual network interface can have performance close to unprotected messages in a single-user machine. To show the performance of the direct VNI, Table 4-4 details the cost of sending and receiving messages in FUGU at kernel level and at user level using two different atomicity mechanisms. The cycle counts are made from simulator traces of a simple ping-pong benchmark and the timings have been verified against the hardware.

The send side of the direct VNI always uses hardware. The send cost of seven cycles shown in Table 4-4 corresponds to a null message sent via a blocking `inject` operation. The receive cost is given two ways, corresponding to reception via an interrupt and reception via polling loop. The interrupt-based receive cost represents the basic fast path cost. The interrupt path includes two trips through the operating system plus the minimal time for a null handler. The polling cost represents a polling loop that receives exactly one type of message. The loop checks the message type by testing the handler address. This sort of polling loop is useful in applications that orchestrate communication closely.

The atomicity mechanism and GID manipulations are performed in software in the current system ("soft atomicity" in Table 4-4). However, Table 4-4 also includes predictions of the performance expected using the revocable interrupt disable mechanism by eliminating the appropriate categories. The result is that the overhead of the fast path for user-to-user communication in FUGU is comparable to the overhead for unprotected, kernel-to-kernel communication.

4.2 Buffered Path

The buffered path allows undeliverable messages to be received and stored in order to preserve the semantics of the UDM model in the face of uncommon but unpredictable cases. The objective of the buffered path is to provide identical semantics to the fast path using memory for data access and user thread manipulations for atomicity control. The objective is achieved using a combination of hardware and software mechanisms and software conventions.

This section discusses the mechanics of buffering and reports the performance of the implementation on on microbenchmarks. For the purposes of this section, the buffer is assumed to be unlimited in size. The means of accomplishing that illusion (virtual buffering) are described in Chapter 5. The

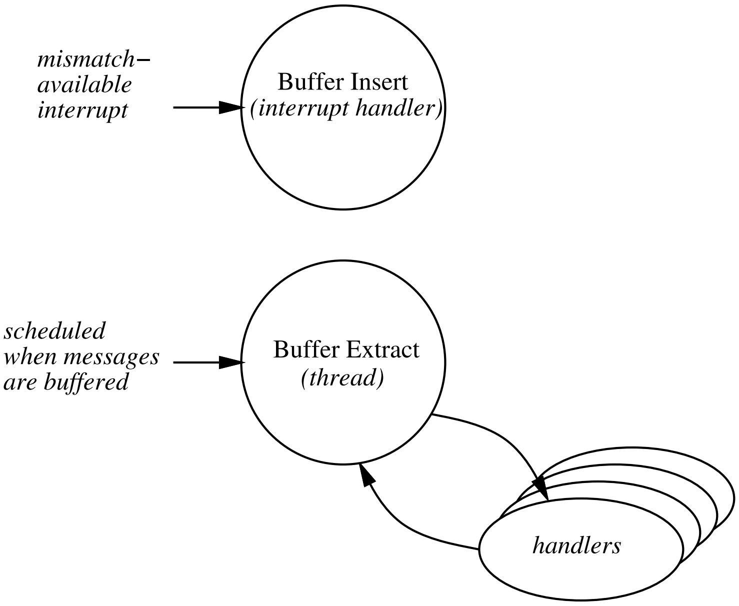


Figure 4-5. Software buffering uses two components: buffer insertion code that runs as the *mismatch-available* interrupt handler and buffer extraction code which runs as an independent thread. The buffer extraction code runs in a loop the invokes the appropriate user handler for each buffered message as long as there are messages.

performance of buffering in real applications is deferred to Section 7.4

Buffering Mechanics. Switching to the buffered case serves as a uniform response to all situations where fast case delivery is not possible or not sufficiently timely. Buffering is a per-process *mode*. In the buffered-mode steady state, the operating system stores messages in a software buffer in the virtual memory of the application performing the communication and the application reads the messages from the software buffer as if from the network interface. A process remains in buffered mode until the last buffered message is handled. On exit from buffered mode, the operating system reverts to allowing user messages to be received directly from the network interface.

Buffering mode uses operating system software with hardware support in the form of the *divert-mode* bit in the network interface (Figure 4-3). The software consists of two pieces depicted in Figure 4-5: buffer insertion code that runs as an interrupt handler and buffer extraction code that runs as a thread. The *divert-mode* bit supports both buffer insertion and buffer extraction. The steady state of buffering mode is illustrated as a timeline in Figure 4-2.

Buffer insertion is accomplished by the *message-available* interrupt handler. When *divert-mode* is set, all incoming messages cause kernel *mismatch-available* interrupts. The *mismatch-available* interrupt handler in the operating system (top in Figure 4-5) demultiplexes incoming messages into the software buffer of the application indicated by the GID in the message header. The DMA option on the `dispose` instruction is used to efficiently copy the message to memory.

Buffer extraction makes use of a thread (bottom in Figure 4-5) that loops, calling the handler for each message, as long as there are message in the queue. The *divert-mode* bit supports buffer extraction as well. *Divert-mode* set causes the user-mode `dispose` instruction (in the user handler)

Item	Cycles
Minimum buffer-insert handler	180
Maximum handler (w/`vmalloc`)	3,162
Execute null handler from buffer	52
Overhead for an isolated null message	about 1400

Table 4-5. Cycle counts for overhead to insert and extract messages from the software buffer. Add roughly 4.5 cycles per argument word to the extraction cost for non-null messages.

to take the *dispose-extend* trap. The *dispose-extend* trap handler then emulates the disposal of a message in the software buffer of the current application. In our current implementation, queued messages are always processed in order.

The buffered delivery mode presents the user with the same atomicity semantics as the fast path hardware by a combination of buffer management and thread priority manipulation. First, if software buffering was invoked because of a timeout or page fault in an atomic section, the thread scheduler defers handling subsequently buffered messages until the suspended atomic section completes, preserving atomicity. Second, handler execution is made atomic in buffered mode by elevating the priority of the message-handling thread so that it always runs in preference to other background threads. If the application was in the midst of a handler or polling for messages at the time buffering was invoked, then that handler or polling thread becomes the high-priority, message-handling thread and can continue to run, reading messages from the software buffer, as long as it keeps atomicity on. Alternatively, if there is no such existing thread or the existing thread exits its atomic section (as detected by *atomicity-extend*), then a new message-handling thread is created to run the handlers of the messages remaining in the buffer.

Buffering Performance. Buffering adds a performance cost when used. The buffered path introduces two components of overhead over the fast path. First, there is an extra copy operation: an operating system handler must copy the message from the network interface to memory. Second, the user handler must now retrieve the message from main memory DRAM rather than from the faster network interface SRAM. For message handlers that run for a long time, the extra overhead of buffering will be insignificant. For short handlers or for messages with large amounts of data, the extra overhead can dramatically increase the total processor (or DMA) cycles consumed by the message. Any extra overhead is important, even to applications where message latency is not a concern, because the cost of copying represents wasted cycles, and total handler overhead strictly limits the maximum observable messaging rate, as observed in Section 7.4.2.

The implementation of the buffered path is evaluated using a microbenchmark that causes many messages to be buffered. The overheads, including allocation of virtual memory on demand, are tabulated in Table 4-5, listing the minimum and maximum buffer insertion times and the buffer extraction overhead. The minimum overhead per message is 232 (= 180 + 52) cycles, or about 2.7 times the fast path overhead of 87 cycles, for null messages. For non-null messages, the difference increases due to the extra cost of pulling the messages from DRAM of 2 cycles per word plus 10 cycles per 4 words for cache misses. The buffer insertion handler uses DMA to copy the message so there is no direct overhead to the processor for extra words inserted into the buffer. The null handler time already includes the cost of one expected cache miss for fetching the message header. The virtual buffering scheme allocates page frames from the operating system on demand. These allocations are expensive (3,162 cycles), but occur so rarely as to be negligible in our simulations.

4.3 Transparent Access

The key to two-case delivery is that the direct and buffered modes of operation must appear identical to user software. This principle of *transparent access* must apply to both the messaging and atomicity primitives of the UDM model. Given transparent access, the runtime system is free to switch to and from buffered mode at any time. As a consequence, the buffered mode provides a unified means of dealing with all exceptional circumstances that prevent a user-level application from proceeding immediately.

Transparency Mechanisms. Transparency of the messaging primitives is achieved through a combination of hardware mechanisms and software conventions. First, `inject` operations are always directed at hardware queues. As a consequence, only the `extract` and atomicity manipulation operations must be virtualized.

As discussed in Section 4.1, an `extract` operation is decomposed into memory-mapped reads from the network interface, followed by a `dispose` instruction. Access to receive data is made transparent by employing a software convention of using a known base register to point to the input message buffer. In the fast case, this register points at the hardware queue. When delivery must be shifted from fast to buffered mode, the base register is altered to point to the buffered copy of the message (if any) in main memory. The `dispose` instruction is made transparent by causing it to be trapped and emulated whenever the the user is in buffered mode.

Atomicity control is made transparent by switching seamlessly between modes. In the fast mode, the atomicity bits control message interrupts directly. In the buffered mode (*divert-mode* set), all interrupts are diverted to the operating system and atomicity is emulated by manipulating user threads as described in Section 4.2.

Mode Transition. Transitions to buffered mode take place when the user cannot or will not make forward progress. In FUGU, there are three reasons to switch the active task from fast to buffered mode (these are demanded for protection and context-switching): page faults in the handler, atomicity timeouts, and scheduler quantum expirations. All three of these events are “soft” in that they merely cause a transparent switch to buffered mode. The user observes an increase in the cost of messaging, but no change in program semantics.

Transparency is important at the beginning of a scheduler quantum, since it allows the scheduler to start a user thread in buffered mode, letting the thread process messages that were received while other threads were scheduled. When the buffered messages have been exhausted, transparency can again be invoked to switch back to the fast mode of reception.

4.4 Discussion

The two-case delivery technique raises a number of research questions. The major question is whether two-case delivery can provide a performance benefit over a system that simply always buffers messages. We have shown in this chapter (Table 4-4) that the implementation of two-case delivery in FUGU achieves low-level message overheads near those of unprotected hardware. The primary experiment in Chapter 7 will show that fast-case performance is achievable over a useful range of mixed, multiprogrammed workloads expected in a scalable workstation.

Two-case delivery raises several other miscellaneous but interesting questions that are not further pursued in this thesis. Three of these questions are discussed briefly below: whether to use upcalls or to buffer on mismatched messages, how the message handling timeout should be tuned for performance and how the two-case delivery buffering system might be integrated with an application-specific buffering system. Finally, some of the design decisions in FUGU might be somewhat different if the relative costs of operations were different.

Upcall on Mismatch. FUGU offers both upcall-based and buffer-based modes of delivery of messages that arrive for the wrong application. It is unclear which mode, if either, is better in general. Upcalls give the lowest overhead and certainly the lowest latency for the application receiving the upcall. Low latency may be crucial for user-level shared memory implementations. On the other hand, buffering probably gives the least disruption to the application that happens to be running when the mismatched message arrives. The buffer insertion handler is known to be fast and to have a limited cache and TLB footprint.

Handler Timeout. The timeout on handler/atomic section execution time is a free parameter that the operating system can adjust as required for performance. Since the difference between direct and buffered modes is transparent to the application, the operating system can cause a switch to buffered mode at any time. There might be different timeouts for each application or different timeouts for same-domain and cross-domain upcalls. Changing the timeout is a scheduling issue: long timeouts are presumably good for the application that is receiving messages. Short timeouts are good for the rest of the system because the short timeout will limit the amount of congestion in the network or the amount of time taken away from an interrupted application.

Further, the timeout might be useful as a means of introducing buffer to benefit even a single application. Mukherjee notes in [58] that judicious use of buffering can improve the performance of some applications running standalone by reducing sender stall time and network congestion. The timeout in message handlers in FUGU could be used as a heuristic for detecting opportunities to improve performance through buffering. Other cues such as whether the input hardware FIFO is full or hybrid heuristics are possible. The ramifications of the timeout are not examined in this thesis.

Buffering Systems. Programmers using active messages apparently often build ad-hoc buffering systems. It is not clear the extent to which two-case delivery can obviate the need for ad-hoc solutions. Alternatively, if the application benefits from some kind of specialized buffering system, it might be useful to merge the system's two-case delivery with the applications. The exokernel-based implementation of buffering in FUGU is amenable to such specialization but we have not tried to take advantage of it.

As a separate buffering issue, the software buffer cleanup thread currently executes the handler for every buffered message before returning to direct delivery mode. This policy allows us to preserve the ordering of messages in the network. However, this policy can also cause the system to remain in buffered mode if the send rate exceeds the receive rate even if the receiver is making progress. An alternative policy would be to forgo ordering and process direct and buffered messages alternately, using the atomicity timeout to limit the time spend in the handlers of buffered messages.[1]

Relative Costs. The buffer enqueue handler overhead is important to system performance, as will be shown in detail in Chapter 7. The minimum buffer enqueue handler overhead of 180 cycles listed in Table 4-5 includes a user interrupt time. In another implementation of two-case delivery, the relative costs of operations could be quite different. Here are the basic tradeoffs which we will revisit in Chapter 7:

- Cross-domain upcalls in FUGU are only slightly more expensive than ordinary user interrupt upcalls because of a tagged TLB and the fact that Sparcle's multiple register sets can be protected from one another. More expensive domain crossings would make buffering on mismatches significantly more attractive.
- The buffer enqueue handler time could be improved by tail-polling in the interrupt handler, or by downloading into the kernel the most common case of the buffer enqueue code. If a small amount of extra hardware were to be invested in FUGU, a good use would be a limited automatic DMA function to accelerate the most common case of buffer enqueue.
- The thread scheduling overheads in FUGU are abysmal, as revealed in the cost to receive an isolated message via the buffered path (Table 4-5). Support for threads could be made as fast as 10s of cycles (as in the "featherweight" threads proposed in [40]). With fast threads, the cost a single message handled on the buffered path would be dominated by the cost of servicing the inevitable cache miss.

Applicability. Two-case delivery is the primary architectural technique that gives the direct VNI performance given the UDM model and the requirements for protection. Two-case delivery is usable in any system with a direct interface. However, two-case delivery is particularly attractive if the system can also guaranteed delivery. Guaranteed delivery allows the fast path to avoid *all* buffer management overhead (pointer manipulation and acknowledgment messages). Guaranteed delivery requires a reliable network and unlimited buffering. While some systems have provided effectively unlimited buffering by providing large amounts of physical buffering (*e.g.,* the SP-2 [72]), pinning down physical memory is inconvenient in general, particularly in a multiprogrammed system where it is desirable to be able to guarantee some minimum amount of buffering to each application.

This chapter has described the first architectural technique, two-case delivery, used to support the direct VNI. The next chapter describes the second architectural technique, virtual buffering, that maintains the illusion of unlimited buffer space while requiring only limited amounts of physical memory in practice.

[1] As yet another separate buffering issue, we could buffer at the sender as well at the receiver with nearly the same hardware. The required addition would be a sender-side divert mode bit which would cause `launch` instructions to trap. The policy decisions of when to turn sender-side buffering on and off are unclear.

Chapter 5

Virtual Buffering Technique

Two-case buffering, detailed in the previous chapter, achieves much of the triple goal of programmability, protection and performance for the direct virtual network interface. Virtual buffering, the topic of this chapter, is a complementary architectural technique that provides two-case delivery with an effectively unlimited buffer at the receiver. The buffer is made to be "effectively unlimited" in size by storing buffered messages in pages of virtual memory allocated on demand. The maximum buffer space available to a process is limited only by the swap space available to the process. At the same time, the buffer is physically "small" from the perspective of the operating system because the buffer is demand allocated and pageable. Despite the large, guaranteed buffer of the programming model, the amount of physical memory required can be minimal.

The benefits of virtual buffering correspond closely to the challenges that arise in the design of the virtual buffering system architecture. The benefits and challenges are described here along with the general architecture of the virtual buffering system. The rest of the chapter is devoted to examining each of the challenges in detail. As introduced in Chapter 2, virtual buffering offers three primary benefits over a system that buffers messages in a fixed amount of physical memory:

1. **Unlimited Buffering.** Virtual buffering aids the programmability of the direct VNI by implementing the unlimited buffer space feature of the UDM model. The existence of a very large buffer space helps programs avoid deadlock: a protocol will not deadlock due to a lack of resources in the network.

2. **User Flow Control.** Virtual buffering allows improved application performance by allowing the application to specialize away flow control (and buffer management) protocol overheads. Providing unlimited buffering at the receiver removes the need for a buffer management or flow control protocol in the message system in the common, direct case. Buffer management overhead is incurred only in the uncommon, buffered case and flow control is provided, explicitly or implicitly, by the UDM user. Virtual buffering thus permits improved performance of the direct VNI compared to a system with a small, fixed-size buffer that must incur buffer-related costs on every message.

3. **Virtualized Resources.** Virtual buffering allows improved performance of the system as a whole by decoupling the physical resources used for buffering from the logical resources. The operating system is allowed to manage (and presumably to minimize) the physical resources

dynamically.

In short, the existence of an effectively unlimited buffer at the receiver both aides the programmer and improves the performance of the fast-case, hardware part of the message system. The premise of virtual buffering, like two-case delivery, is that demand for buffering is rare: good programs and ordinary conditions naturally keep buffering requirements low and the operating system is at liberty to devote memory resources to other purposes. The costs and complexity of virtual buffering are then confined to the support for rare cases of unusual programs and operating conditions where performance is less of a concern.

The challenges in the design of the virtual buffering system are associated with each of the three benefits:

1. Unlimited buffering aides the programmer in avoiding deadlock, but the virtual buffering system must be constructed in such a way that it does not, itself, deadlock when it runs low on physical memory. Since virtual buffering tends to be invoked when the network is filled with messages, some mechanism is required to assure a path to backing store. The direct VNI makes use of a reserved, second, logical network to support virtual buffering, but we find that the performance requirement for this second network is low.

2. User flow control is a benefit to applications because it potentially allows the application to depend on implicit or customized limits on message generation rather than suffer the costs of flow control solely for the sake of the low-level message system. However, the virtual buffering system must contend with applications that fail to limit messages and therefore fail to limit their buffering demands, temporarily, erroneously or deliberately. The direct VNI includes an *overflow control* mechanism that applies a coarse form of flow control on such applications via the scheduler. Overflow control reduces the demand for buffering in marginal programs or, at worst, prevents runaway programs from slowing down the rest of the machine.

3. Decoupling virtual buffer space from physical storage introduces the need for the operating system to automatically manage the storage resources and to reconcile the storage demands of virtual buffering with other consumers of physical memory such as virtual memory. We have explored resource management only minimally, other than the resource management implicit in overflow control.

The virtual buffering system consists of several software components that address the three challenges. Figure 5-1 depicts the components as an expanded version of the generic software buffering system illustrated in Figure 4-5 of the previous chapter. Most actions are initiated by the buffer insertion and extraction routines, as in any software buffering scheme. The virtual buffer insert handler must occasionally allocate fresh physical pages for storage, (A). The allocator and the extract code interact with the memory manager, (B), to maintain the supply of fresh pages. A combination of insert/extract events and memory supply conditions trigger the overflow control module, (C), which uses the scheduler to modulate the flow of messages indirectly.

The actions of the various components of the virtual buffering system are illustrated in Figure 5-2. In the common case, (1) messages are consumed immediately by the application via the fast case and no part of the buffering system is involved. In the usual buffering case, (2), messages are buffered in memory, using the buffer insert/extract modules and, when necessary, the allocator (Figure 5-1A)

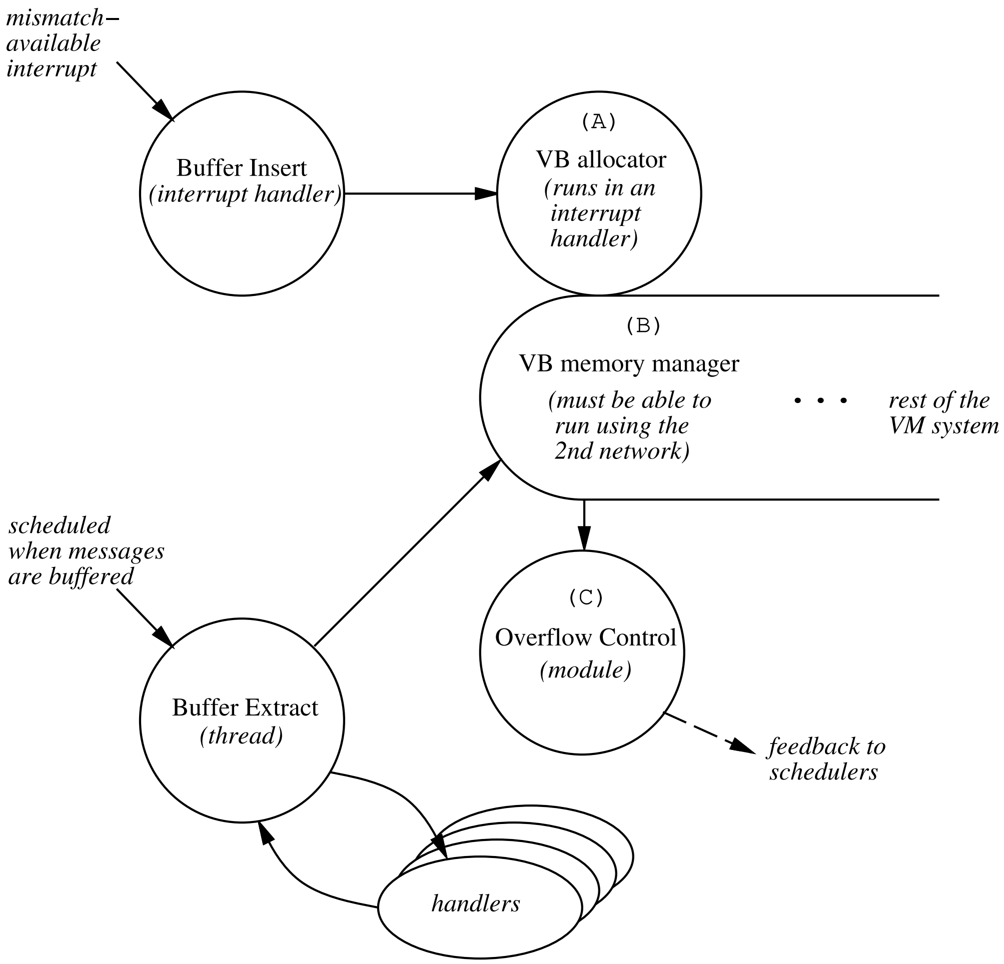


Figure 5-1. Virtual buffering (VB) makes use of three software modules in addition to the buffer insertion and extraction modules that are common to any software buffering scheme. The virtual buffering buffer-insert handler must be able to allocate fresh pages quickly (A) and to perform paging if necessary (B) without using the main network. The memory manager, triggered by buffer insertions and extractions, interacts with the overflow control module (C) to detect and throttle applications that appear to be temporarily or permanently out of control.

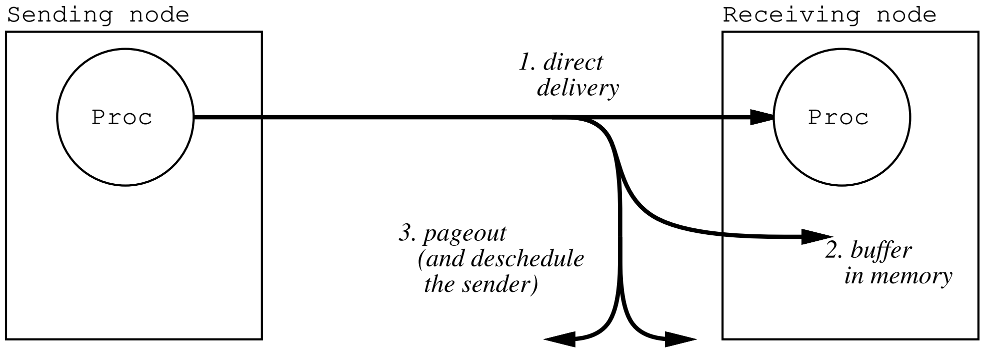


Figure 5-2. There are three message reception cases. In the fast case, (1), messages are received directly by the processor. In the ordinary buffering case, (2), messages are buffered by software in memory, but the memory is physical memory that is resident on the node. In the worst case, (3), a node can continue to receive messages while simultaneously paging out message buffers. A reserved second network is used to assure that paging can proceed without deadlock in the worst case as well as for the overflow control mechanism that helps avoid paging.

and memory manager (B) modules. When physical memory runs low, (3), the second network serves as a guaranteed path for paging by the memory manager, (B) if required. The second network also serves as a path for control messages passed by overflow control, (C), and used to throttle the sender.

The three challenges arise in various parts of the buffering system. The deadlock avoidance issue arises in all modules that may be invoked to insert a message, *i.e.,* the buffer insert code and modules (A), (B) and (C) in Figure 5-1. All must be able to continue to operate while the main network is blocked. User flow control is addressed by the overflow control mechanism and policy implemented in the overflow control module, (C). Resource management policy is the domain of the virtual buffering memory manager, (B), in combination with the rest of the virtual memory system.

The prototype direct VNI we have implemented includes all parts of the virtual buffering system except that the memory manager, (B), is limited. The memory manager supports allocation and participates in overflow control but does not perform paging. We discuss the requirements for paging in this chapter, but more subtle aspects of memory management are not well developed.

The remainder of the chapter discusses each of the three virtual buffering design challenges in turn. Section 5.1 discusses the support for unlimited buffering using a reserved second network to avoid deadlock. Section 5.2 describes overflow control used to preserve system performance under high buffer demand and faulty user flow control. Section 5.3 describes the simple resource management policy used in the prototype and discusses the issues that would arise in a more complete system. Finally, Section 5.4 wraps up with a discussion of tradeoffs and of research questions for future work.

5.1 Unlimited Buffering

The unlimited buffering feature of UDM and the direct VNI aids the application in avoiding deadlock. Therefore, for the sake of irony as well as for correctness, it is crucial that the virtual buffering system must not, itself, deadlock. The buffer must be able to accept new messages under all circumstances, including when physical memory is low and the main network is blocked. Further, since buffering is uncommon but does still occur regularly, it is important for the buffering system to operate as efficiently as possible. For instance, memory allocation should not ordinarily require message traffic or an expensive thread switch.

The fact that the buffer system must accept messages under all circumstances has two consequences. First, all the code potentially invoked to buffer a message must be able to run under restricted conditions. The buffer insertion interrupt handler (Figure 5-1) that inserts messages into a virtual buffer usually just adds the message to several already buffered on an existing physical page. If necessary, the handler invokes the operating system to quickly allocate a fresh physical page to extend its buffer. The allocation decision is made locally and the allocator in fact runs in the interrupt handler. Second, in order to continue buffering messages to the limits imposed by the swap space, the buffer system requires a guaranteed ability to transfer pages from physical memory to backing store. The ability to transfer pages out allows the memory system to recycle those physical pages and continue buffering without deadlock.

Second Network. For the direct VNI, we choose to supply a second network as the guaranteed path to backing store. The second network is used infrequently for this purpose so its performance is not critical. The network might be shared with some other use, such as supporting shared memory. An extra virtual channel [16] in the main network, a LAN or a service network could serve as an adequate second network. Our emulator hardware provides a custom but very simple, bit-serial network.

The virtual memory system in our prototype system handles demand allocation, but does not perform paging, so paging via the second network has not been implemented. The key services in the virtual buffering path (*e.g.,* , paging) would use a communication abstraction that uses the main network when the main network is clear, but otherwise resorts to the second network.

We considered and discarded two alternatives to a second network for guaranteeing the ability to page. First, we could have require a paging device be attached to every node in the machine. This requirement limits the machine configuration as well as the amount of swap space available to any one node. A second network is much more flexible and can also carry other miscellaneous memory management traffic beyond just the bulk data that needs to be moved to backing store. Second, we could guarantee that the operating system could use the main network by introducing a mechanism to globally shut off user traffic before physical memory resources on any node of the machine drop below a critical level. While such an approach may be practical, the bounding the amount of space and time required is difficult.

The purpose of providing unlimited buffering is as a deadlock avoidance technique. Other high-performance anti-deadlock schemes make use of a second logical network as well. For instance, the common request/reply discipline makes two logical networks explicit. In contrast to request/reply, the two networks in virtual buffering are highly asymmetrical: the second network exists to guarantee a solution but is rarely used. Also, the strict gang scheduler in the CM-5 uses a separate control network to initiate gang switches for protected multiprogramming. The direct VNI's second network is used similarly to provide protection, but only on demand.

Performance. The performance of the message system degrades gracefully as demand for buffering increases. Referring to Figure 5-2, in the fast, common case, (1), the message system runs at hardware speeds. Message reception overhead in the prototype is 115 cycles for a null message (from Table 4-4). When buffering is invoked, but physical space is available for buffer storage, every

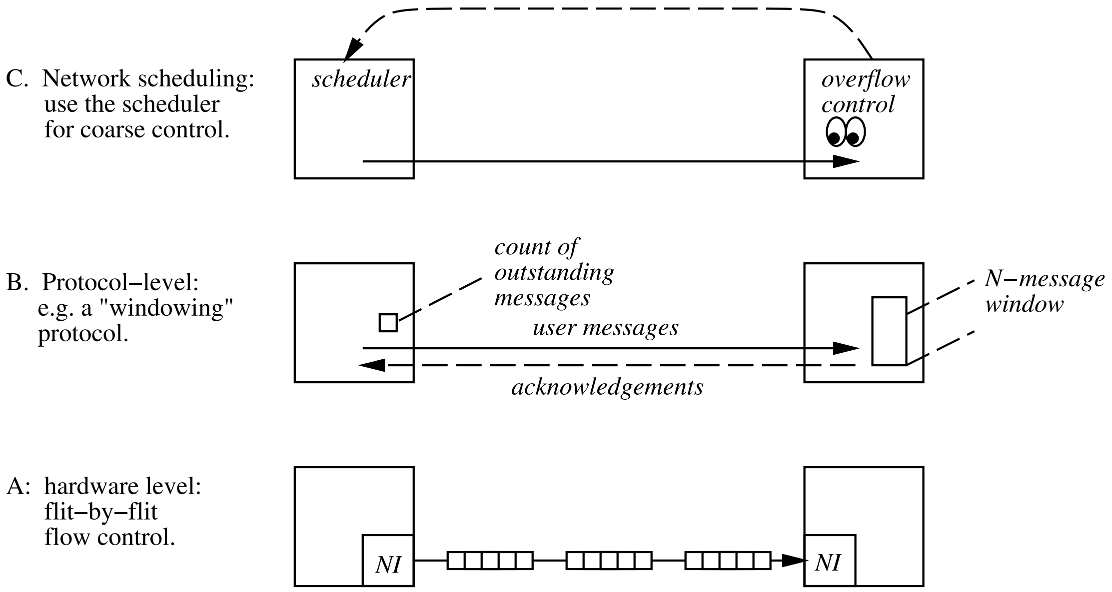


Figure 5-3. Flow control is applied at different levels of the message system for different purposes. At the hardware level, (A), flow control is applied flit-by-flit to avoid dropping the data of individual messages. At the protocol level, (B), a message protocol may count the number of outstanding messages to limit resource consumption. An explicit layer of protocol may not be necessary if flow control is implicit in the characteristics of the application. Finally, overflow control, (C), is an indirect technique, a form of "network scheduling", that has the effect of applying flow control at a coarse grain.

message must pass through memory, (2), at a reduced rate. In the best case (no allocations and messages are handled in batches), the overhead per message received is about 232 cycles for a null message (the sum of 180 cycles insertion time and 52 cycles extract time from Table 4-5).

A fresh page allocated and mapped by the operating system costs 3,162 cycles. Allocations are rare both because many small messages will fit on a page and because the buffer system caches a few free pages in a local free page list rather than re-allocating every time. Finally, when the system is in the mode of paging and accepting messages simultaneously, the rate of incoming message data it limited to the rate of outgoing pages, *i.e.,* trivially:

$$BW_{in} = BW_{out} \tag{5.1}$$

This demand on the second network isn't as formidable as it may seem at first glance for two reasons. First, the messages coming in are likely to be be small messages so that BW_{in} is less than the primary network's full bandwidth. In contrast, paging traffic on the second network is in the form of bulk messages so that BW_{out} can be a high fraction of the achievable bandwidth of the second network. Second, and more importantly, this demand on the second network would only be decisive if the system were to run in this mode for extended periods. The purpose of overflow control, described in the next section, is to avoid the paging mode. Overflow control also uses the second network, but depends on the second network's *latency* rather than on its bandwidth.

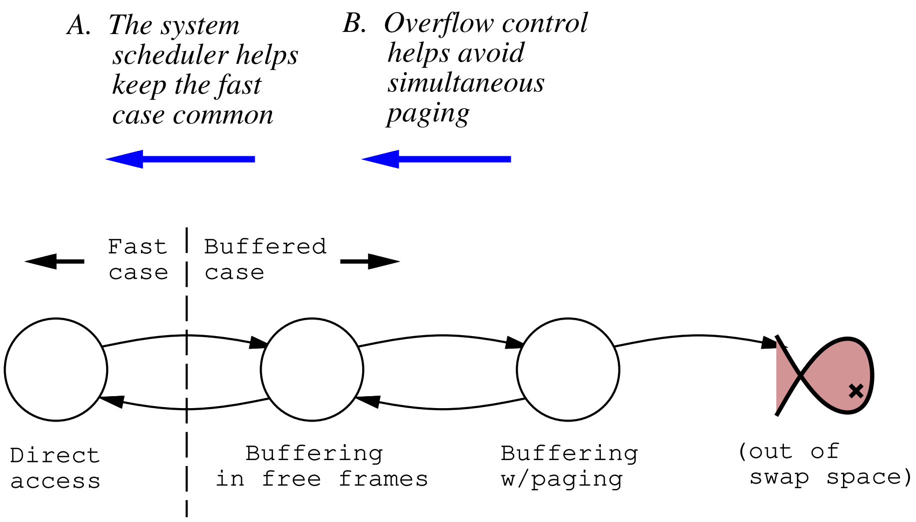


Figure 5-4. Virtual buffering system operating modes with software oversight. The system scheduler is expected to help keep the fast case common by favoring coscheduling of applications that communicate intensely. The overflow control mechanism works to slow applications that appear to consume large amounts of buffer space.

5.2 User Flow Control

Virtual buffering gives the application control over protocol-level flow control. Such an arrangement is beneficial because much of the time the application may need *no* additional flow control beyond what is implicit in the algorithm. In essence, the application is given the ability to rely on the hardware-level flow control for best performance in the common case. Because virtual buffering provides unlimited buffering, however, the system must also include the means to limit the effects of individual applications that demand excessive buffering.

Flow control can be implemented at different levels in a message system, as illustrated in Figure 5-3. At the bottom, (A), the hardware (in a reliable network) provides flit-by-flit backpressure so that no messages flits are dropped. At the middle level, (B), message systems often provide a flow control protocol using extra processing and messages to limit the number of possible outstanding messages. The protocol serves the purpose of dealing with a limited buffer. In the direct VNI, we forgo the middle-level flow control protocol, leaving that to the user, and add a top-level "overflow control", (C), based on minimally invasive measurements and coarse feedback through the scheduling system.

The concept of using system software to indirectly "oversee" the operation of the application pervades the direct VNI message system. Figure 5-4 depicts the modes of the virtual buffering system (as originally shown in Figure 3-5) with the addition of the two entities used to oversee the smooth operation of the system. First, the system scheduler, (A), is expected to coschedule applications that benefit from coscheduling and thereby reduce the demand for buffering. The scheduler has the effect of "pushing" the system from the buffered cases to the direct-access case. Second, the purpose of the overflow control system, (B), is to keep the virtual buffering system from paging.

The second network provides a guarantee of deadlock avoidance, but performance would degrade severely if we were to routinely block the main network while paging. In practice, high consumption

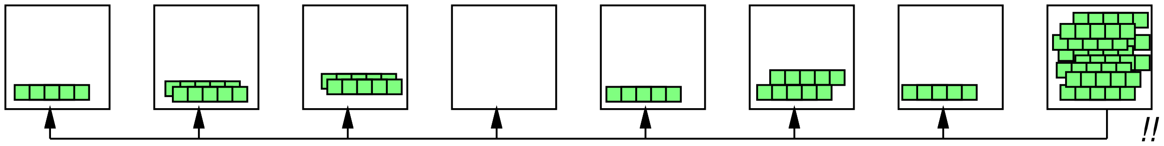


Figure 5-5. Overflow control mechanics. The buffer insertion code for each process monitors the amount of buffer space used compared to the physical memory resources available on a node. When demand for buffering is high, one node (here the rightmost), switches the whole application into overflow control mode using messages sent on the second network.

of virtual buffering space corresponds to severe misscheduling of an otherwise reasonable application or to an application that is maliciously or erroneously out of control. Excessive demand for virtual buffering in our system is analogous to thrashing of virtual memory. Accordingly, we employ a technique reminiscent of the anti-thrashing strategy in Unix: we identify the offending application and take gross control of its scheduling. First, an application on the verge of exhausting physical memory is globally suspended while paging clears out space on the node. Second, a well-behaved application will recover from buffering if gang scheduled (Section 7.4.2), so the buffering system advises the scheduler to gang schedule the application.

Overflow Control. The system that detects excessive demand for buffering and feeds that information back to the scheduler is called overflow control. Overflow control has much the same effect as a very coarse-grain flow control mechanism. We have implemented one overflow control mechanism and policy using the second network to deliver messages with low latency given that the main network is likely to be blocked.

The overflow control mechanism stops all the processes in an application when one process detects an excess of messages. Figure 5-5 illustrates the stopping action. When any one process observes a potential problem, it causes all other processes in the application to halt temporarily. The overflow control mechanism uses the second network to deliver "stop" messages to all processes in a job once the buffer space in any one process exceeds a threshold and the number of free pages on a node falls below a low-water mark. On reception of such a stop message, each affected process switches to an "overflow" mode in which it only consumes messages (using a heuristic based on our knowledge of which threads do what in our system). All processes in the job remain in overflow mode until message buffers in the job have shrunk to zero size or the number of free pages on the node rises above a high-water mark. When the job is released from overflow mode, all processes in the job return to normal mode.

Figure 5-6 depicts the paging system high and low water marks and Figure 5-7 shows the threshold in the message buffer. Note that the high and low water mark in the paging system represent an ordinary implementation of virtual memory management [74]. The switch from normal to overflow mode and back again is then governed by two equations:

$$\text{Normal} \Rightarrow \text{Overflow} \;\; \textbf{if} \;\; \exists_{processes} \left((L_{queue} > L_{threshold}) \wedge (N_{pages} < N_{lowwater}) \right) \tag{5.2}$$

$$\text{Normal} \Leftarrow \text{Overflow} \;\; \textbf{if} \;\; \forall_{processes} \left((L_{queue} = 0) \vee (N_{pages} > N_{highwater}) \right) \tag{5.3}$$

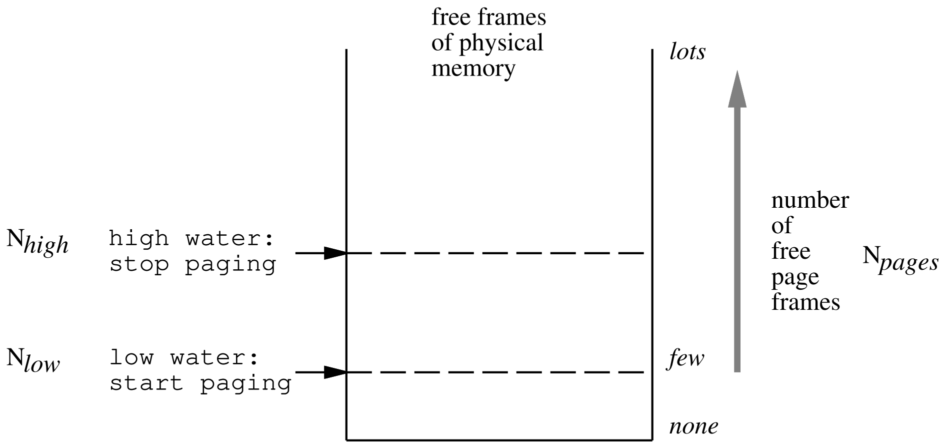


Figure 5-6. In a conventional paging system, the pager starts when the number of free frames on a processing node falls below a low water mark and stops when the number of free frames exceeds a high water mark.

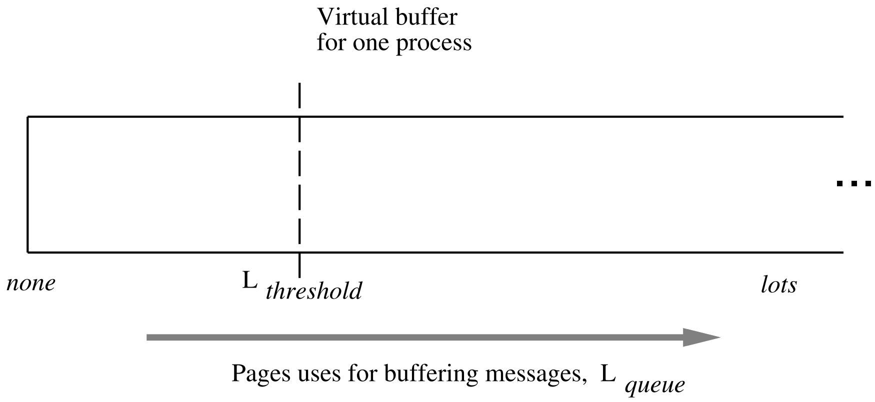


Figure 5-7. Overflow control is triggered by a combination of the size of the virtual buffer in use by one process and the amount of free physical memory available on a processing node. The threshold is set in terms of pages of physical memory used for buffering.

The overflow control policy assumes that either the application will clear its messages or that the system pager will free up pages to the high-water mark. The former is characteristic of a well-behaved application. The latter has the effect of turning runaway memory consumption into a slowdown. The slowdown affects only the application using buffering since overflow control system reacts before the node runs out of free pages.

Performance. Overflow control uses the second network to pass its control messages. The main demand on the second network arises because the latency of overflow control messages affects the amount of reserved space required on the node to avoid paging. In detail, when a node decides it is time to apply overflow control, it sends out overflow control messages. However, in the interval between when the overflow messages are launched and when the messages are received, some additional messages will be injected into the network. The worst case arises if the network is running at full bandwidth:

$$M_{required} = T_{latency} \cdot BW_{in} + M_{network} \tag{5.4}$$

where $M_{required}$ is the amount of reserve memory required, $T_{latency}$ is the latency of an overflow control request, both in hardware and in software (presumably mostly software), BW_{in} is the message data bandwidth into a node as before and $M_{network}$ is the amount of storage in buffers of the network fabric and in network interface hardware. The network buffers must be counted because the receiving node still needs to be able to empty the network after the overflow control messages have stopped the senders.

The best policy for overflow control remains an open question. In our implementation, overflow control stops and later restarts all the processes in an application together on the assumption that the whole application will either correct itself or should be treated as permanently misbehaved. A complimentary mechanism (not implemented) is that the occurrence of an episode of overflow control should be treated by the scheduler as a suggestion to coschedule that application. Lee [43] experimented with stopping one process (just the apparent sending process) of an application. Such an approach might be able to help automatically manage the scheduling of threads within an application for the benefit of the application.

5.3 Resource Management

Virtual buffering gives the operating system the opportunity to manage physical resources used for buffering. Given that freedom, the operating system then needs a policy for allocating and deallocating physical page frames used for virtual buffering. This policy needs to coexist with other consumers of page frames in the operating system, *i.e.,* the virtual memory manager and the file cache manager.

We have not proposed or evaluated any such policies, other than the resource management implied by overflow control. One interesting question is what information is needed to best optimize buffer usage. For instance, message rates could be measured as well as simple buffer sizes. A second interesting question is how virtual buffer management should interact other demands for physical memory, *i.e.,* virtual memory and file caching. The page frame cache could be partitioned or unified. A third interesting question is what kind of API one might export to take advantage of application "advice".

5.4 Discussion

The virtual buffering technique raises several research questions. Beyond basic feasibility, the major question is whether, having given the user the offer of "unlimited" buffering, the demand for real buffering is in fact overwhelming. The second experiment in Chapter 7 (Section 7.4) will show that demand for buffering remains low in our sample applications, even when buffering is artificially induced. The sample applications provide adequate flow control implicitly due to their intrinsic characteristics. While flow control can always be added in a user-level library, we will show that the conditions for avoiding excessive buffering in the direct VNI are quite mild. Further, the overflow control mechanism described above in Section 5.2 is evaluated in Section 7.5, where we show that it can effectively throttle runaway applications, to their benefit and to the benefit of the rest of the system.

In addition to the major question, three research questions were raised above in this chapter and only partly addressed. First, the second network, used for deadlock avoidance and as part of overflow control, is intended to be used rarely and thus as a component has only a second-order impact on performance. However, the exact performance requirement of the second network in a real system is unknown. The bandwidth and latency issues described in Sections 5.1 and 5.2 are described in additional detail by Lee [43]. Second, the domain of overflow control mechanisms and policies is broad; the scheme described in Section 5.2 is only one simple possibility. Third, as mentioned in Section 5.3, we have not addressed the resource management questions in virtual buffering beyond the use of our overflow control policy.

Cross-Domain Messages. Another issue that is only minimally addressed in the direct VNI is the question of cross-domain messages. Cross-domain messages are incompatible with the idea of unlimited buffering. The problem is that one application can flood another application's swap space with messages, which implies that the applications must trust one another. There are three possible approaches to reconciling virtual buffering with cross-domain communication between mutually distrustful applications:

- The first approach is to allow cross-domain communication with unlimited buffering but to store the buffered messages in swap space charged to the sending application.
- The second approach is to abandon unlimited buffering for cross-domain messages and instead provide some other guarantee, such as a fixed-size buffer negotiated at the time the cross-domain connection is set up. Protocols built atop cross-domain messages would the have to explicitly take the buffer size limit into account.
- The third approach, taken in the direct VNI architecture, is just to outlaw cross-domain messages. Either the server or the client must be distributed (even if only as a "stub") so that inter-process communication is always local to a processor. This organization is actually quite natural for a scalable workstation in which all the kernels are trusted. For instance, a distributed server such as the web server used for the experiments in Chapter 7, can optimize its internal communication and perform caching transparently. The server interface abstraction is decoupled from the server's communication requirements, which is a useful property.

Applicability. Since the mechanisms of virtual buffering are non-trivial, it is convenient to be able to implement the mechanisms in software. Using virtual memory is particularly natural when the processor initiates all the buffering because existing support for virtual memory (*e.g.,* the processor's TLB) is reused. It requires a relatively complex DMA engine or coprocessor to manipulate virtual memory independently [26, 57, 66, 80].

However, virtual buffering is usable in any system that employs buffering. For instance, a system that performs limited buffering in hardware could implement virtual buffering by using interrupts to dynamically expand the buffers [56, 57, 80]. One simple way to perform limited buffering would be to give the hardware a small table of active page frames indexed by the GID. The hardware would perform buffer insertion into these pages automatically until the page frame was full and then cause an interrupt. The hardware methods have all the same software complexity issues as our all-software method, albeit with a less demanding time constraint since traps occur less often. The interrupt handler that responds to a hardware buffer overflow must still be able to allocate a page frame in an unusual situation and the system must still address deadlock, overflow control and resource management.

The previous chapter and this chapter have described the architectural techniques of two-case delivery and virtual buffering used to support programmability, protection and performance in the direct VNI. These chapters have been oriented to the architecture, although we have covered many of the implementation details as concrete examples. The next chapter describes the experimental system in which the direct VNI was implemented and evaluated.

Chapter 6

Experimental System

The direct virtual network interface architecture described in the previous two chapters introduces several new ideas that demand detailed evaluation. We chose to evaluate the ideas using an experimental hardware and software system ("FUGU"). This chapter describes the details of the experimental system, its capabilities and the limitations required to make its construction tractable.

Systems that require customized hardware and a customized operating system are particularly difficult to evaluate in a research environment with limited resources. An ideal evaluation system would include all hardware features operating at full speed, a full operating system implementation and multiple real-world applications running simultaneously. The construction of an evaluation system for research purposes requires compromises.

Figure 6-1 categorizes three possible compromise approaches used in systems research. All three approaches work by building something more tractable than the actual target system, and then bridging the difference with simulation and analysis:

- The first approach (6-1A) is to build a limited version of the system, concentrating on a key component and evaluating its performance in detail. The hope is that the performance of the key component can be extrapolated to predict the performance of the full system.
- The second approach (6-1B) is to build a variant of the full software system on stock hardware and attempt to account for the differences in hardware and software separately. One drawback here is that much conventional operating systems work is oriented to *accommodating* stock hardware so the point of the experiment may be lost: rather than exhibiting a combined hardware/software solution, the focus shifts to how cleverly one can work around limitations of the stock hardware.
- The third approach (6-1C), taken in FUGU, is to build the full system using hardware that can be easily modified. The resulting "emulated" system is accurate in many senses but runs slowly, speed being the typical trade-off for easy mutability.

We performed our direct VNI experiments on two platforms. First, we constructed an emulator of FUGU hardware by adding a small amount of additional hardware to an existing experimental multiprocessor, Alewife [1]. Second, we built a custom simulator of the system, T2. The two

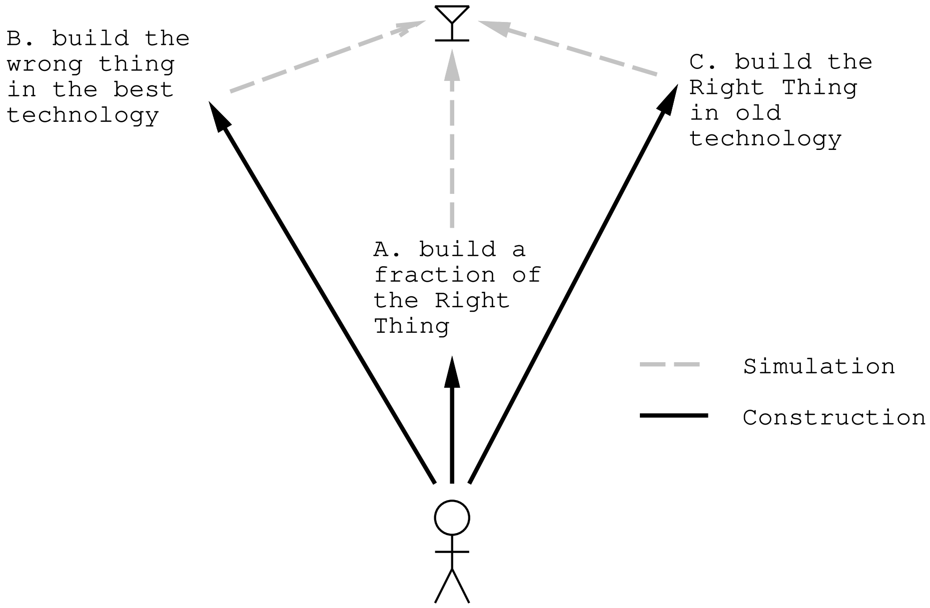


Figure 6-1. Three compromise approaches to mixed hardware-software system evaluation: (A) build a limited version of the right thing, (B) build something different but related on stock hardware, (C) build a full version on emulated (slow) hardware.

platforms are described in Section 6.1. The bulk of the direct VNI system is implemented in operating system software. The FUGU operating system is based on Exokernel techniques [20]. Glaze is the kernel part of the OS and PhOS is the library part of the OS. The FUGU system also makes use of an external scheduler that runs as a separate application, microkernel-style. Section 6.2 describes the operating system and the scheduler. Finally, Section 6.3 describes communication libraries used atop the UDM messaging model.

6.1 Hardware

The FUGU hardware is implemented two ways: in the form of emulated hardware and in the form of a fast simulator. The hardware is an "emulator" in the sense that it is based on a modification of an obsolete (ca. 1993) processor set. However, the base machine, Alewife, has the virtue of being a complete, balanced implementation for its generation. The conclusions about the message system should continue to be applicable to current and future machine generations. The simulator is a fast, instruction-level simulator with support for instruction-driven (as opposed to true structural) timing models. The timing in the simulator is loosely calibrated to match the hardware.

6.1.1 Emulated Hardware

The FUGU hardware is constructed using the chipset of the Alewife machine and extending it with a User Communication Unit (UCU) component, implemented as an FPGA. Figure 6-2 shows the block diagram for a (1-processor) node with the UCU component shaded. Figure 6-3 shows a photograph

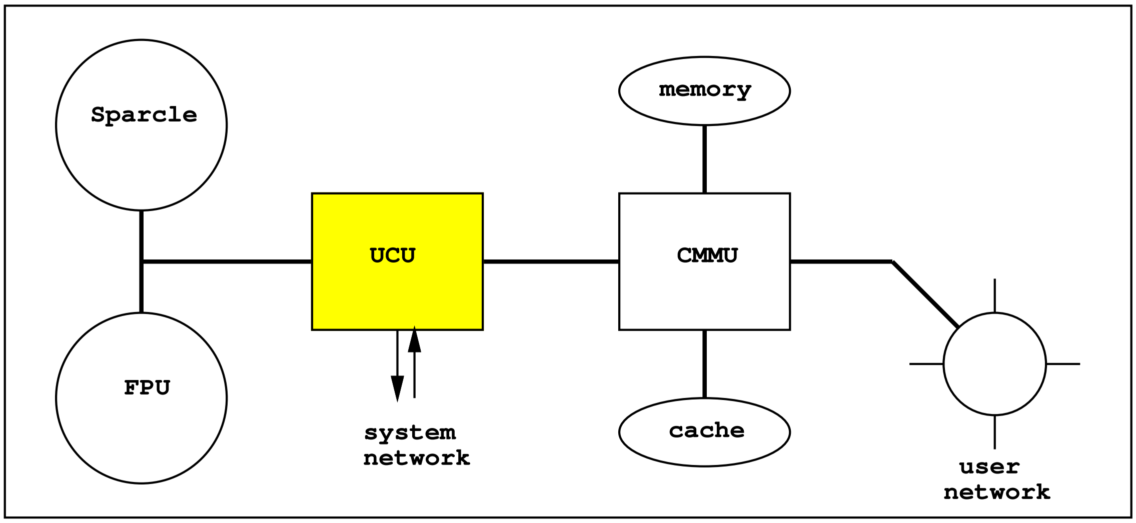


Figure 6-2. Each FUGU node contains a processor ("Sparcle"), an FPU coprocessor, a User Communication Unit (UCU), a Communication and Memory Management Unit (CMMU), a one-level cache, DRAM and a network component. The white components are from the Alewife chipset; the UCU component (shaded) was the only additional hardware required for FUGU.

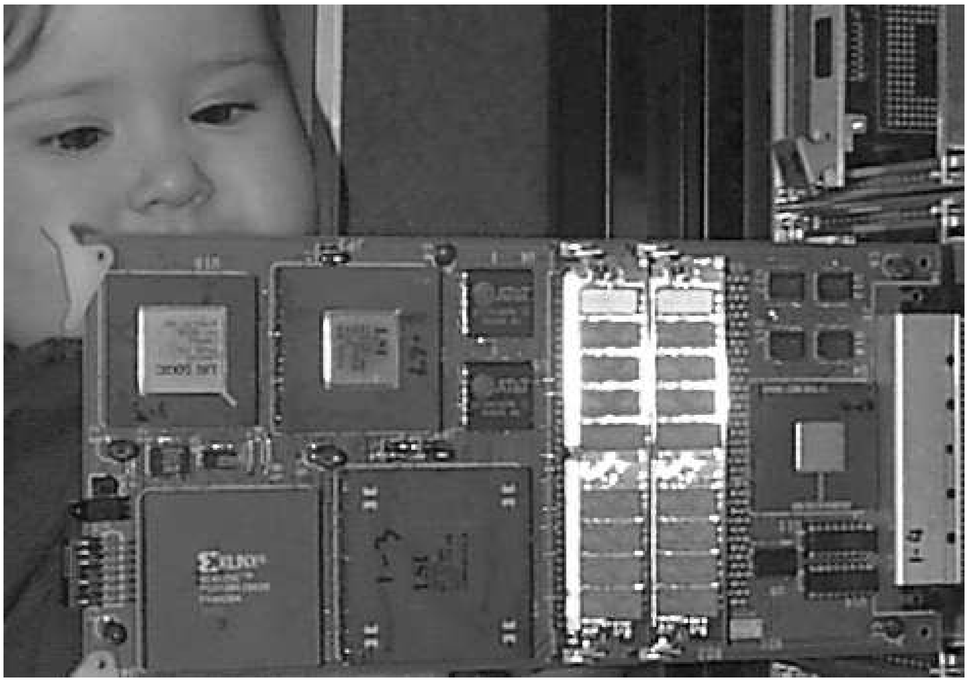


Figure 6-3. A FUGU node is implemented as a single 4.66" by 9.91" PCB. Multiple nodes are connected together via a point-to-point network embedded in a passive backplane. The four major ICs at the left are, clockwise from top left, the FPU, the Sparcle IPU, the CMMU and the UCU. The major IC at the right is the network component, a Caltech Mesh Routing Chip (MRC).

of one node. The node PCB measures 4.66” by 9.91”.

The Alewife chipset includes a Sparcle (SPARC V7-derived) processor, a stock SPARC floating-point unit, 64KB of direct-mapped cache organized as 16-byte lines, 32MB of DRAM, a Caltech Elko-series Mesh Routing Chip (EMRC) and a Communication and Memory Management Unit (CMMU) gate array. The CMMU component implements shared memory, the message passing interface and the revocable interrupt disable mechanism as well as acting as the cache controller, DRAM controller and network buffer.

The machine reuses much of the Alewife infrastructure beyond the chipset: the FUGU nodes are connected together by placing them in a corner of an unused Alewife backplane, FUGU is connected to a host workstation via the same boards and cables used for Alewife and the communication server program running on the workstation is a modified version of the corresponding program used for Alewife.

Alewife is a single-user, physically-addressed machine while FUGU supports multiprogramming and virtual memory. In a machine built from the ground up, such support would be designed into each relevant component: the TLB in the processor, protection checks in the network interface, etc. For FUGU, new protection and virtualization features are added externally to the Alewife chipset in the UCU FPGA.[1] The UCU component extends the Alewife chipset with three main features:

1. A translation cache (TLB) to support virtual memory. The TLB is 4-way associative with 128 entries. It supports user-level probe operations to support translation for DMA [55].
2. A GID-check mechanism intended to be used during polling (not tested).
3. A rudimentary second network. The second network is a 1-bit, token-passing ring network [43].

The direct VNI approach allows the network interface hardware to remain quite small. The network interface consumes about 20K gates plus 640 bytes in compiled SRAM arrays in about 0.40mm^2 of the CMMU, an LSI 300K-series gate array. That’s roughly 20KB of cache SRAM in the same technology.[2]

The implementation of FUGU using an add-on FPGA to an existing machine represents a compromise approach in two ways. First, the system is slow compared to the machines we ultimately want to represent.[3] The Alewife chipset alone is capable of running at 20MHz; the external FPGA added to the critical address path makes it even slower. The projected maximum system clock speed is 7MHz, although at the time of writing it has not been tested above 2.5MHz. Second, several protection features proved fundamentally difficult to implement in the UCU and are left unprotected in the prototype. These features could be trivially implemented correctly if we were building a machine from scratch. There are three such features: the GID stamp is left to user code; the hardware GID check on polling relies on a particular access idiom in user code (the header must be touched first); DMA translation uses an unprotected TLB probe operation followed by a user-level write to the send- or receive-descriptor.[4]

[1] Jon Michelson [55] and Victor Lee [43] designed and implemented the UCU FPGA. Victor also built the PCB.

[2] The size estimate is based on the relative areas of the network interface to the cache tags array in the CMMU. The cache tags are implemented as a compiled SRAM array, so the comparison is highly unfavorable to the network interface, which is dominated by the area of random gate array logic.

[3] 300MHz processors are common in 1997.

[4] Heinlein describes a scheme for safely passing both virtual and physical addresses to an NI in [27]

A two-node FUGU machine exists running a subset of our applications. We used the machine to run the microbenchmarks behind Tables 4-4 and 4-5 and for gross calibration of the simulator against applications. Most of the results in Chapter 7 come from the simulator.

6.1.2 Fast Simulator

Most of the results come from a fast simulator, T2, used for system development.[5] The simulator uses dynamic compilation to achieve speed and flexibility. The simulator design emphasizes speed and employs an "incremental" approach to timing accuracy [4]. Compared to measurements of the emulated hardware, the simulator reports cycle counts within +0/-30%. We believe the distortions introduced do not qualitatively affect the results.

The simulator models timing using an instruction-driven approach. The simulator does not model detailed timing of, for instance, pipelines in the processor, but instead assigns an execution time to each instruction based on conditions at the time the instruction is executed. The simulator includes models for the processor, the memory hierarchy, the network and filesystem I/O:

- The processor model includes static integer instruction times but not pipeline stalls. Floating point instruction timing is not modeled.
- The memory model includes the timing of cache misses to local memory. The default parameters mirror the hardware: the hierarchy is single-level using a 64KB, direct-mapped, unified cache with 16-byte lines and a 10-cycle miss penalty. Shared memory is included but the applications used for the evaluation in Chapter 7 do not make use of it and its timing is not modeled.
- The network model provides for a fixed wire delay plus structural delays due to buffer limitations and message ordering. We model limited network buffering in two places. First there is a lumped model placed at the receiver representing all the send/network/receive queues along the message path. Second, there is a model for the visible send queue window. Contention inside the network is not modeled.
- I/O in the FUGU system is implemented by having the operating system exchange messages with the host workstation. The host appears as a node in the network that is logically outside the set of processors. I/O messages are roughly at the level of Unix system calls (*e.g.,* open/close/read/write/stat). I/O timing in the simulator is modeled crudely as either a fixed delay or a random, exponentially distributed, delay added to each I/O message. I/O time is not significant in any of the applications, so we use a small, randomized delay solely to vary timings across multiple runs.

The fact that the simulator tends to underestimate processor time makes the effects of the network interface relatively more important. The importance of costs we introduce in the message system (*e.g.,* via software buffering) are thus not hidden behind processing overhead.

[5] The simulator was designed and initially implemented by Robert Bedichek.

6.2 System Software

The FUGU system software consists of the operating system, a scheduler and several communication libraries we have ported to the machine. The operating system is an extensively modified version of a research OS and the scheduler is fully custom while the libraries are ported with minimal modifications from other systems. The compiler used is a standard GCC (version 2.7.2.1) for the Sparc. We use the `-mflat` and `-mno-app-regs` flags to suppress the use of register windows and most global registers, respectively, for Sparcle. All parts of the system system and applications are compiled with the optimizer on (`-O3`).

6.2.1 Operating System

The FUGU operating system is a custom multiuser operating system organized as an Exokernel [32]. As an exokernel, the OS consists of two pieces: "Glaze" is the in-kernel portion and "PhOS" is the library operating system portion. Glaze and PhOS are directly derived from the original Aegis/ExOS exokernel [20] and reuse much of that code.

Glaze and PhOS support preemptive multiprogramming, virtual memory, user-level threads and a low-cost interprocess communication (IPC) mechanism. The virtual memory implementation is limited in that there is no paging to disk. However, it does use page faults to allocate and zero-fill pages on demand and to propagate pages of shared memory on demand. These page faults introduce the situation of messages rendered undeliverable due to the memory system.

Glaze and PhOS implement the software side of two-case delivery and virtual buffering. The two-case delivery implementation switches to buffering mode in response to page faults, a certain DMA case (described in Appendix A) and optionally in response to mismatches. Message timeouts are currently fatal. The timeout value is set to a large value for debugging purposes.

As an exokernel, both *message-available* and *mismatch-available* interrupts are propagated from Glaze to PhOS as user interrupts. User interrupts in Glaze use a reserved pair of register windows in the Sparcle processor to accelerate saving and restoring registers. The *mismatch-available* user interrupts are implemented as cross-domain upcalls involving a context switch. The context switch is relatively inexpensive because of the tagged TLB in the UCU and because the register window support is still usable across protection domains.

There are two consequences of the design decision to make *mismatch-available* interrupts into cross-domain upcalls. First, the virtual buffer enqueue handler always involves the full cost of a user interrupt. As we will show in Chapter 7, the buffer enqueue throughput is particularly crucial to the system so in retrospect this design decision was flawed. The second, positive, consequence is that it is easy for PhOS to support multiple buffering policies. In Chapter 7, we make use of upcall-always, buffer-on-mismatch and, for illustration, buffer-always policies. Since the policy decision is made in PhOS, not Glaze (or in hardware), it is easy and natural to change policies on a per-application basis for experimental purposes.

Several versions of overflow control are implemented in PhOS, including the version integrated with the virtual memory system, as described in Chapter 5 and a simpler version used for experiments in Chapter 7. A third version was implemented by Lee for [43]. Since overflow control is implemented in PhOS, the mechanism is partly cooperative. As with most resource management

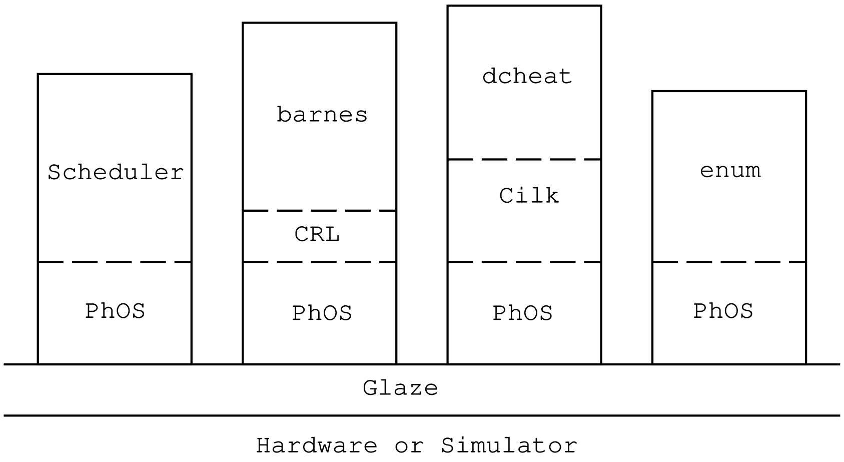


Figure 6-4. A single FUGU node supports multiple applications running on an Exokernel-based operating system. Glaze is the kernel part of the exokernel; PhOS is the library part, which is shared by all applications. The system scheduler runs as a user application in microkernel fashion.

mechanisms in Exokernels, the cooperative mechanisms must ultimately be backed by some coercive mechanism in the kernel. We have not implemented a kernel version of overflow control.

Figure 6-4 illustrates how the various software components are used together. Glaze is the kernel part of the Exokernel and PhOS is the library part. We use only a single library OS linked with all the applications although PhOS supports some convenient per-application specializations such as the ability to change the message buffering policy for experimental purposes. CRL and Cilk are additional communication and load balancing libraries, respectively, used by some applications described in Section 7.1. The scheduler, described below, is implemented as an ordinary application. In particular, it has access to the same communication facilities as other applications, which is convenient because the direct VNI functionality is *not* available to the kernel. As an application, the scheduler is unique only in that it has permission to control CPU allocation.

Figure 6-5 depicts the assignment of applications to processors for one of the experiments in Chapter 7 Each box represents a distinct protection domain. The flock of Web Robots and the Web Server, described in Chapter 7, form an interactive, client-server application. Barnes, also described in Chapter 7, is a fine-grain scientific application using the CRL library. The Sequencer is a short-running, sequential application used to select and run the main applications for the experiment. Finally, the Scheduler and Propagator (a part of the scheduler), described next, are parts of the global runtime system.

6.2.2 Scheduler

The FUGU scheduler is Walter Lee's experimental "flexible coscheduling" scheduler that supports both conventional scheduling based on priority queues and coordinated (gang or co-) scheduling based on synchronized clocks [44]. The desired scheduling mode is applied per-application. The flexibility in the scheduler allows us to explore the tradeoffs that two-case delivery permits. Ideally, the scheduler would switch between scheduling modes autonomously and dynamically based on its evaluation of the effect of the tradeoff for each application and for the system as a whole. For our

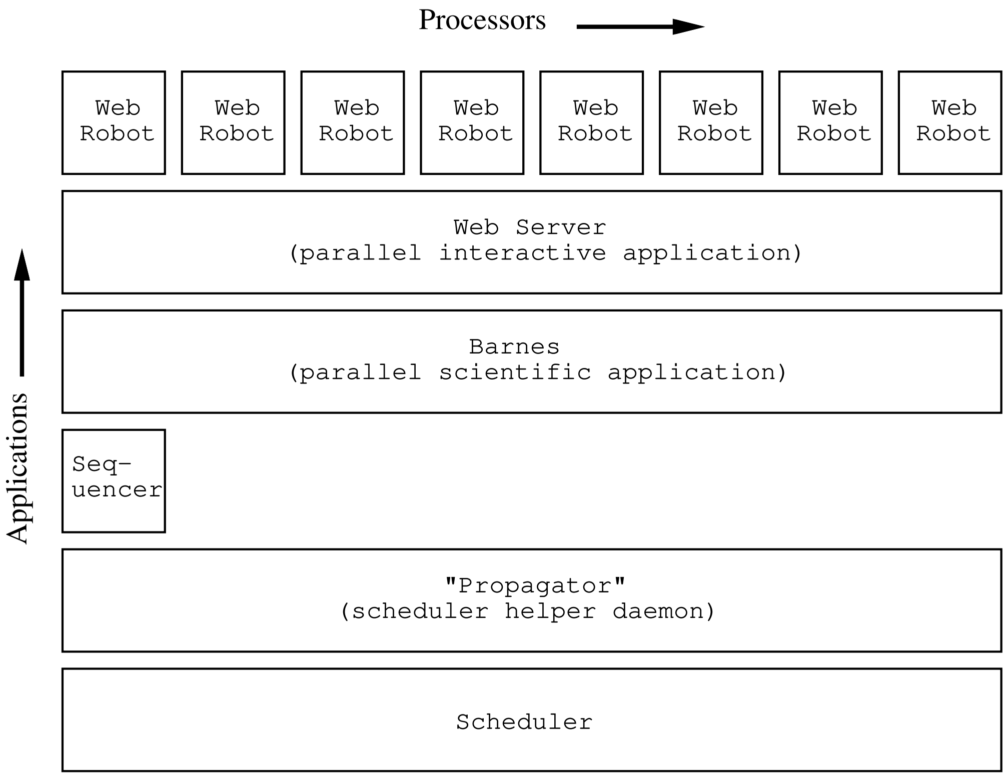


Figure 6-5. A FUGU system runs a mixed workload of multiple parallel and sequential applications distributed across the machine. Here the x axis represents the eight processors in this machine while the y axis represents process "slots" on each processor. The workload represents a typical set of applications for one of the experiments in Chapter 7. Four applications, including the scheduler, are parallel applications conceptually running on eight virtual processors while others are sequential.

purposes, we set the mode of the scheduler manually for each application in each experimental run.

The scheduler utilizes the notion of a scheduling "round". A round is several timeslices-worth of pre-planned activity. The scheduler pre-plans the execution of ganged jobs in a round but merely reserves space for non-ganged jobs in the round. The use of a round allows the scheduler to amortize the cost of computing and communicating the gang schedule over several time slices. The cost of using the round is a decrease in responsiveness for scheduling ganged jobs.

For ease of development, we implemented the scheduler as a separate, user-level server program in micro-kernel style (Figure 6-4). The scheduler program is distinguished from other applications only in that it owns all the "CPU slices" exported by Glaze. Thus, at the end of each timeslice, Glaze transfers control to the scheduler since the scheduler always owns the next CPU slice. The scheduler then directly yields to the application it chooses to run.

6.3 Libraries

The low-level UDM communication model described in Chapter 3 is intended to serve as a building block for more programmer-centric communication abstractions. We have used three communication models with FUGU. One is a minimal abstraction atop the UDM primitives and is defined in this section. The second is the CRL software shared memory system [30] ported to use UDM. The third (unused in the experiments) is the Cilk language and its associated load-balancing runtime system [5].

The minimal abstraction atop UDM (conventionally referred to as just "UDM"), wraps the `inject` and `extract` operations in `do_on` statements and the abstraction of "handler" declarations for procedures, respectively. The `do_on`/handler abstraction implemented for FUGU is strictly interrupt-driven.

A `do_on` statement looks like this:

`do_on` (*dst, handlername [, arg0 [, arg1 [, ... [, arg13]]]]*)

The *dst* argument is an integer representing a virtual processor number and *handlername* is a pointer to the handler procedure to be invoked on that virtual processor. The arguments, *arg0, arg1,* etc., are arbitrary items that are passed by value (the value must fit into a single, 32-bit word). The `do_on` statement simply compiles into a table lookup to translate the `dst` node number into a header followed by an `inject` operation.

A handler declaration looks like this:

```
void handler (handlername [, arg0 [, arg1 [, ... [, arg13]]]])
{
   ⟨atomic handler code⟩
   user_active_global();
   ⟨thread code⟩
}
```

The handler declaration compiles into a procedure and an initial stub containing the `extract` operation to load the procedure arguments (if any) into registers. The handler procedure is invoked automatically as a user interrupt and runs with atomicity on (interrupts disabled). The handler optionally includes a `user_active_global()` statement, which converts the handler into a thread and then invokes the `atomicity(`*off*`)` operation described in Chapter 3.

Since all actual uses of `inject` and `extract` operations are wrapped in the stylized `do_on` and handler forms (even CRL and Cilk are implemented atop `do_on`/handlers), we took some liberties in the implementation of the direct VNI in the name of expediency. These two shortcuts also have the effect of closing an atomicity "hole" between TLB probes and the storing of physical addresses for DMA. The direct VNI description in Chapter 4 describes how to maintain transparency in access to the network interface in the face of arbitrary user instructions.

The first shortcut is in `do_on`. If a context switch occurs in the middle of a partially constructed message, the design chapter prescribes saving and restoring the message output buffer registers (Figure 4-3). Instead, we roll back the current thread's PC to the point in the compiled `do_on` code at which the output buffer registers are first written. The constraint placed by this technique is that the block of code (generated by the compiler) beginning with the first store to an output register and ending with the `launch` instruction must be idempotent.

Second, since `extract` operations in FUGU appear only in handlers, we took the liberty of using a dual-stub-and-rollback approach to transparency instead of emulating the `dispose` instruction in buffering mode as described in Chapter 4. The dual-stub approach works like this: the compiler actually generates two versions of the initial stub containing the `extract` operation, one for direct mode and the other for buffered mode. The direct-mode stub is constructed to be idempotent up until the `dispose` instruction. If the system must switch to buffered mode while a handler is running but before the `dispose` point, the handler is merely restarted using the buffer-mode stub.

Beyond raw UDM, we ported two libraries to the machine. One is CRL [30], an all-software shared-memory system. CRL is strictly a runtime library that operates by passing messages. We use this library extensively in our experiments in the next chapter. The other is Cilk [5], a programming language and runtime system that performs automatic load balancing through work stealing. This version of Cilk (version 4) communicates through shared memory. Since the focus of the thesis is on message-passing, we do not include any Cilk-based applications in our evaluation. The next chapter introduces the experimental evaluation and presents the results.

Chapter 7

Results

The previous chapters have shown that direct virtual network interface provides programmability through the UDM model, is compatible with protection for multiuser operation and provides low-level performance near that of unprotected hardware. This chapter turns to the performance of the direct VNI with real applications, alone and under multiprogrammed conditions. The central hypothesis is that the performance benefits of a direct interface persist over a useful range of mixed workload conditions expected in a scalable workstation. The chapter describes the test applications used to form a mixed workload and then proceeds to make two main points in response to the questions raised in the design chapters, Chapters 4 and 5:

1. The first and most important point is that the "optimistic" approach to performance of two-case delivery remains justified for a broad range of workload conditions and with an achievable set of system parameters. Using a mixed workload of an interactive application running against compute-intensive parallel applications, we observe that only $14 - 33\%$ of messages are buffered in our parallel applications while 10% of the CPU time is devoted to interactive work.
2. The second point is that the logically unbounded buffers provided by virtual buffering do *not* lead to unbounded real buffer consumption. "Well-behaved" applications naturally limit their demand for buffering and ill-behaved applications can be throttled by overflow control.

Before introducing the applications and mixed workload experiments, the *potential* benefits of the direct VNI approach in terms of programmability, protection and performance should be clear from the previous chapters:

- Programmability comes from the UDM model described in Section 3.1. UDM provides kernel-like control over interrupts and the ability to treat the network as private, reliable and with unbounded buffering. UDM is powerful enough to implement the efficient Active Messages [78] and Remote Queues [8] models directly.
- Protection is enabled because the combination of two-case delivery and virtual buffering solves the undeliverable message problem (Section 2.2), permitting protected multiprogramming and demand-paged virtual memory.

- Performance in terms of speed comes from the direct interface. Microbenchmark results presented in Chapter 4 (Table 4-4) show that the base message-passing costs in FUGU are comparable to the costs of an unprotected, direct interface on a single-user machine built on the same hardware base. Others have shown that tightly-coupled, direct interfaces tend to be more efficient than indirect, memory-based interfaces [28, 56]. Thus FUGU's *peak* performance is high.

Two-case delivery and virtual buffering provide good system performance in terms other than speed as well. Virtual buffering enables low memory consumption compared to a system that must provide a fixed amount of physical buffering per application. The operating system is at liberty to deallocate or page out message buffers that appear to be rarely used. Performing buffering in software helps keep the direct VNI hardware small by limiting its functionality to input/output FIFOs and a simple DMA engine. The processor's address translation hardware is reused and all policy complexity is pushed into software. The FUGU NI hardware occupies the area of only about 20KB of L1 cache made in the same technology.

Section 7.1 introduces the set of parallel and interactive applications used to evaluate FUGU in the remainder of the chapter. We review how these applications benefit from the direct VNI when running standalone. The focus of the evaluation, however, is to determine whether the possible high performance is in fact achieved with real applications in multiprogrammed workloads. As mentioned above, there are two main questions.

The first question is whether optimism in message delivery is justified in the presence of interactive applications. Section 7.2 introduces an experiment to generate a realistic, mixed interactive/compute-intensive workload under controlled conditions. The real applications are run together in this mixed workload. The results show that FUGU offers graceful degradation in parallel performance as interactive demands increase because the majority of messages are still delivered along the fast path. This property allows FUGU to have better performance than a system that always pays the overhead of buffering or a system that provides multiprogramming only through strict gang scheduling. Section 7.3 compares the direct VNI approach to a network interface that always buffering messages in memory using hardware support. The comparison is based on a model of the costs of messaging and measurements from the real applications. We find that software buffering is justified for an achievable range of system parameters.

The second question is whether the design decision to effectively guarantee buffering can lead to an uncontrollable demand for buffering. The last two sections approach this question from two directions. Section 7.4 shows that ordinary, well-behaved applications naturally limit their demand for buffering. Through an experiment that artificially induces buffering in a controlled manner, the demand for buffering is observed to increase gracefully in our applications as buffering is induced. A synthetic application is used to define the meaning of "well-behaved". Finally, Section 7.5 evaluates one overflow control policy for controlling buffer consumption in an ill-behaved application. The section shows that buffer consumption in an ill-behaved application can be effectively converted into a slowdown for that application alone.

App.	Description	Data set	Model
Barnes	N-body simulation	2048 bodies, 4 iterations	CRL
Water	Particle-in-cell simulation	512 molecules, 4 iterations	CRL
LU	Blocked matrix decomposition	256x256 matrix, 16x16 blocks	CRL
Enum	Triangle puzzle	6 pegs/side	UDM
Barrier	10000 barriers	–	–
Null	Busy wait	–	–
Webserver	Parallel, caching web server	242 documents (774MB)	UDM
Webrobot	Synthetic client (uniprocessor)	1000 random requests	–

Table 7-1. Application descriptions. Barnes, Water, LU and Enum are real, parallel applications. Barrier and Null are synthetic applications included to illustrate extremes in application characteristics. The Webserver and Webrobot applications together form an interactive, client-server application.

App.	Measured	Cycles	Tot. msgs	$T_{\text{interhandler}}$	T_{handler}
Barnes	3rd iter.	45.7M	107,849	3390	337
Water	3rd iter.	47.6M	36,303	10,500	419
LU	all	13.4M	7,564	14,200	478
Enum	all	72.7M	610,148	953	320
Barrier	all	18.5M	240,177	615	149

Table 7-2. Application base measurements (eight processors). $T_{\text{interhandler}}$ represents the average time between messages launched or received, including the time to launch or to receive. T_{handler} represents the average time for message reception, including the interrupt overhead of 115 cycles (all the applications use interrupt-based message delivery).

7.1 Applications and Standalone Performance

The goal of the scalable workstation is provide good performance across a range of mixed workloads. The direct VNI supports mixed workloads by providing a programmable message-passing model with good performance for parallel applications and by supporting protection for multiprogramming with priority-based scheduling for interactive applications. Accordingly, for the evaluation, we develop a set of both parallel and interactive applications that can be combined together in a workload. This section discusses those applications and their performance when running alone on the FUGU system.

The experiments in the remainder of the chapter depend on a set of real and synthetic applications ported to FUGU. The characteristics of the applications used are tabulated in Tables 7-1 and 7-2. Table 7-1 gives general information and Table 7-2 summarizes their characteristics through measurements of selected benchmarks running standalone on an eight-processor system. In the experiments, application running time is taken to be either the execution time of the third iteration for the iterative benchmarks (Barnes and Water) or the execution time of the whole application. The table shows execution time in cycles, the total number of messages, the average time between

# procs	Barnes	Water	LU	Enum
1	254.0	305.0	78.1	713.0
2	148.0	158.0	42.8	306.0
4	83.2	84.1	24.7	142.0
8	47.4	46.4	13.9	75.0
16	26.0	26.4	8.6	37.6
32	16.3	15.5	5.8	21.1

Table 7-3. Application run times in millions of cycles over a range of machine sizes. The run times are measured using the 3rd iteration for iterative applications (Barnes and Water) or the full run time for others.

messages, $T_{\text{interhandler}}$, (derived from the totals) and the average time spent per handler, T_{handler}, (measured separately) for each application.

Compute-Intensive Applications. Three applications, LU, Water and Barnes are standard scientific benchmarks. Water and Barnes are from the SPLASH [71] suite. LU, Water and Barnes are shared-memory programs that have been modified to make use of the CRL all-software shared-memory system [30]. CRL presents a message-passing load that is representative of other coherence protocols such as Stache [62] or Shasta [65] and can be considered operating-system-like: many low-latency protocol packets mixed with larger data packets. Of the three CRL applications, Barnes exhibits the finest granularity in terms of message rate (smallest $T_{\text{interhandler}}$).

A fourth application, `enum`, solves a simple "triangle puzzle" game by exhaustive, breadth-first search [47]. The implementation used here is a fine-grain, data-parallel version that exchanges numerous unacknowledged short messages and synchronizes only infrequently. From Table 7-1, the ratio of T_{handler} to $T_{\text{interhandler}}$ in `enum` is extremely high: a third of the time of the application is spent in handlers. `Enum` is a potentially challenging application for the direct VNI because the combination of a high message rate, substantial computation in handlers and little synchronization can in theory lead to excessive buffering. In practice, however, as Section 7.4 will show, the characteristics of enum are still within the realm of "well-behaved" applications and excessive buffering is not a problem.

The compute-intensive parallel applications obtain competitive speedups on FUGU. Figure 7-1 and Table 7-3 summarize the speedups achieved by the parallel applications. The speedups are relative to the parallel code running on one processor. The speedups for the CRL applications are comparable to those reported in [30] for the Alewife machine. The speedup for `enum` is noticeably superlinear due to cache and TLB capacity misses that decrease with machine size.

Finally for the compute-intensive, parallel applications, `barrier` and `null` are synthetic applications included to illustrate opposite extremes of sensitivity to scheduling. `Barrier` consists entirely of barriers and thus represents a synchronization-intensive application that is extremely sensitive to how it is scheduled. `Barrier` makes essentially no progress unless all its processes are scheduled simultaneously. `Null`, at the other extreme, is an embarrassingly parallel application. Each process in `null` busy-waits for a fixed number of iterations without communication.

Interactive Application. The one interactive application used in the experiments in this chapter

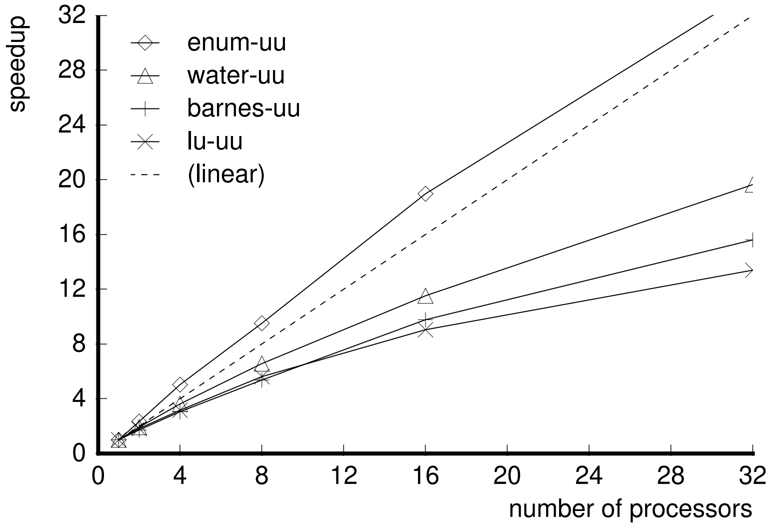


Figure 7-1. Application speedup for applications running standalone. The speedups are relative to the parallel code running on one processor. The speedup for `enum` is superlinear due to cache and TLB capacity effects.

is a client-server system using multiple processes. The system is composed of a single, parallel "webserver" application along with a constellation of sequential client "webrobot" applications.

The webserver listed in Table 7-1 is one interactive, parallel application. The server answers HTTP requests for web pages stored as files on disk. Service is parallelized by using a hash on the web page URL to distribute work among the processors. Pages are cached and in practice the cache fills quickly so that disk activity is a minor part of the run time. The server is always run on all processors. Requests arrive locally via a socket-like interface that uses fast inter-process communication (IPC) supported by Glaze/PhOS between clients and the server.

The webserver is exercised by a synthetic client application, the webrobot. Each robot is a uniprocessor application that generates random requests with a random inter-request time. The requests are selected uniformly from a list of all the known web pages. The inter-request time is an exponentially-distributed random variable with an expected value that is programmable via a parameter to the robot application. The combination of the server and robots makes a realistic parallel interactive application.

Standalone Performance. The direct VNI offers good application performance by giving message system performance close to that achievable with an unprotected network interface on the same hardware. Others have shown that direct interfaces and low-level messaging models give good application performance. We show here that the virtualized interface achieves performance close to a dedicated interface.

Others have shown that tightly-coupled direct interaces and reliable message subsystems offer performance benefits to applications. Henry and Joerg [28] quantified the value of placing the message interface close to the processor and of providing hardware support for buffer management. The CNI work [56] shows the benefit of a tightly-coupled direct interface versus the best current

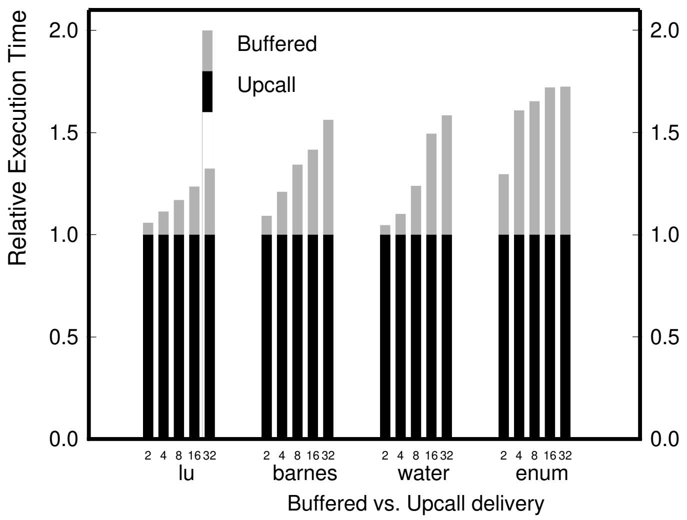


Figure 7-2. Impact of increased message overhead. The gray lines represent the application runtime using a buffer-always policy normalized to the runtime using upcalls. The buffer-always path incurs as much as 1400 cycles of overhead per message while upcalls cost only 115 cycles per message.

memory-based interface with real applications. As discussed in Chapter 2, direct interfaces offer lower latencies for single messages because they eliminate cache miss costs and buffer management overhead. Karamcheti and Chien [33] found that 50-70% of active message overhead in the CM-5 was attributable to the costs of providing flow control, ordering and reliability in software. The uNet paper [77] suggests that buffer management overhead would cost 6uS/message on their Sparc-10-based system. 6uS is considered relatively low extra overhead (10%) in their system but amounts to hundreds of cycles.

We can compare the performance of the direct VNI with that of an unprotected direct interface on the same hardware via the crude technique of looking at the difference in overhead. For instance, according to Table 7-2, a process in an 8-processor configuration running `enum` receives a message every 953 cycles. From Table 4-4, the difference in overhead between our direct VNI implementation and unprotected, Alewife kernel messages on the same hardware is $115 - 54 = 61$ cycles. So, crudely, `enum` could be expected to run about 6% faster if we abandoned protection. The direct VNI implementation could certainly be improved as well, even on the same hardware. From Table 7-2, a implementation using the hardware-supported atomicity mechanism should have a fast path overhead of only 87 cycles. Based on the small differences in overhead, we draw the first conclusion:

> *Conclusion 1: The direct virtual network interface offers standalone application performance close to that of an unprotected direct interface*

The difference between the direct VNI and an unprotected, direct interface is small compared to other possible techniques. If we change the VNI policy to buffer all incoming messages, we end up with a particularly expensive form of memory-based interface. The buffering overhead (232 cycles) in our system and cache miss cost (a 10-cycle penalty) are relatively small. However, the interrupt-through-scheduler path we use for the buffer-buffer policy is very expensive (1000+

cycles). Figure 7-2 shows the relative runtimes of our applications with the buffer-always policy normalized to the runtime with the default (upcall-always) policy. All of our applications perform better under the default policy than under the buffer-always policy, and the effect grows with the number of processors. `Enum` slows down by 65% at eight processors over the direct VNI with the slow message system, compared to the 6% projected gain possible from using unprotected hardware. The buffer-always policy has to be considered a worst case, although it's essentially what SUNMOS [63] does.

This section has shown that the direct VNI offers good performance to applications running standalone. The performance with the (protected) direct VNI is close to that with an unprotected direct interface on the same hardware base. The next section shows that the direct VNI performance benefits persist under multiprogramming using a parameterized mixed workload.

7.2 Mixed Workload Performance

The main result of the thesis is that the direct VNI continues to give good parallel performance under mixed workload conditions. This section and the next section combine to present that result in two ways. Section 7.2.1 starts by describing the mixed workload experiment and how scheduling is performed. Then, Section 7.2.2 makes the first point: we run real applications in that framework and show that, despite multiprogramming, most messages are still received via the fast case. The next section (Section 7.2.2) makes the second point: we examine the tradeoff between the direct VNI and a hardware-supported, memory-based VNI in detail to circumscribe the domain in which hardware-supported buffering can be beneficial and show that the domain is vanishingly small.

A mixed workload is difficult to support because compute-intensive parallel applications and interactive applications have contrary scheduling requirements. Compute-intensive parallel applications, such as the CRL applications in our workload typically benefit from having all processes in the application scheduled simultaneously on all processors for long time slices. Scheduling together allows the component processes to communicate with predictable, low latencies. On the other hand, interactive applications, such as a web server, a database engine, or any application in an I/O-bound phase, benefit from traditional, independent, priority-based scheduling. Independent, priority-based scheduling will give high priority to the interactive application so that its component process or processes can service incoming interactive requests with predictable, low response times.

Reconciling the two demands is an important problem that is unsolved in general. If the interactive demands are sufficiently low, it is clearly useful for the system to employ independent, priority-based scheduling because the disruptions to the compute-intensive application are low. If disruptions are too high, there is emerging agreement [15, 22, 3, 73, 44], that it becomes useful to explicitly co-schedule the processes of the compute-intensive application by one means or another.

Section 7.2.1 introduces a parameterized workload that pits a compute-intensive application against an interactive application. This workload is used to generate a full range of situations and to illustrate the tradeoffs between them. In particular, results from the workload illustrate why some applications should be coscheduled in some situations.

Section 7.2.2 presents results using real applications in this workload running on FUGU. These results confirm the qualitative expectation that most messages arrive via the fast path. First, with low interactive demands, two-case delivery improves performance because most messages arrive via the

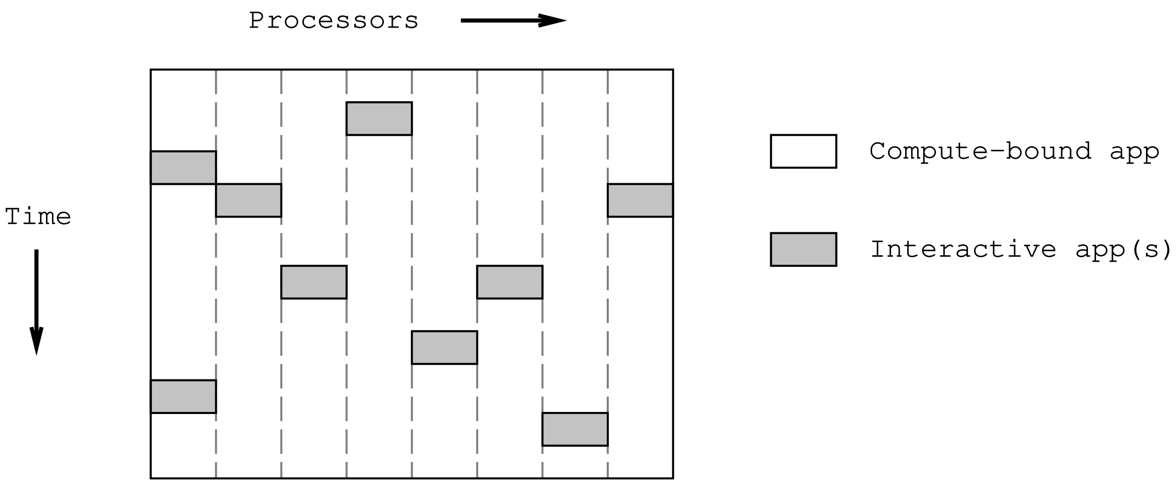


Figure 7-3. In a mixed workload under ideal conditions, the interactive application runs whenever it has work to do and the compute-intensive application runs profitably at all other times. Here, the execution of applications multiprogrammed on an eight-processor machine are represented as a section of a timeline. Time runs from top to bottom and the eight processors are represented from left to right. Shaded blocks represent segments of time in which the interactive application is running on a particular processor.

fast path. Second, at high demand, it is advantageous to switch to coscheduling where, again, most messages arrive via the fast path. As mentioned above, it is often desirable to switch to coscheduling anyway because of application characteristics unrelated to the direct VNI. In the middle ground there is a tradeoff between the two-case delivery approach and an alternate approach that always buffers messages in memory. Section 7.3 goes on to examine this tradeoff in detail.

7.2.1 A Mixed Workload Experiment

This section introduces a framework for evaluating system performance with a mixed workload. We describe the experiment, describe what goes on internally during the experiment, and then show sample graphs illustrating the range of expected results. The basic experiment multiprograms two applications, one compute intensive, and one interactive, on a parallel machine and measure their runtimes. Each application uses all the available processors. Our scheduler (Section 6.2.2) is programmable in two modes. One mode is independent, priority-based scheduling. The other is coscheduling with fixed timeslices.

Consider first what happens with independent, priority-based scheduling. The interactive application immediately and permanently gets the higher priority because the scheduler detects it blocking. Therefore, each interactive process runs whenever it has work to do. Ideally, the interleaving of the two applications in time will look like Figure 7-3. The component processes of the interactive application run at high priority whenever there is work for them to do. The compute-intensive application soaks up all cycles that are unused by the interactive application.

In reality, Figure 7-3 represents a best case. The compute-intensive application is apparently embarrassingly parallel. When processes in a compute-intensive application synchronize with one

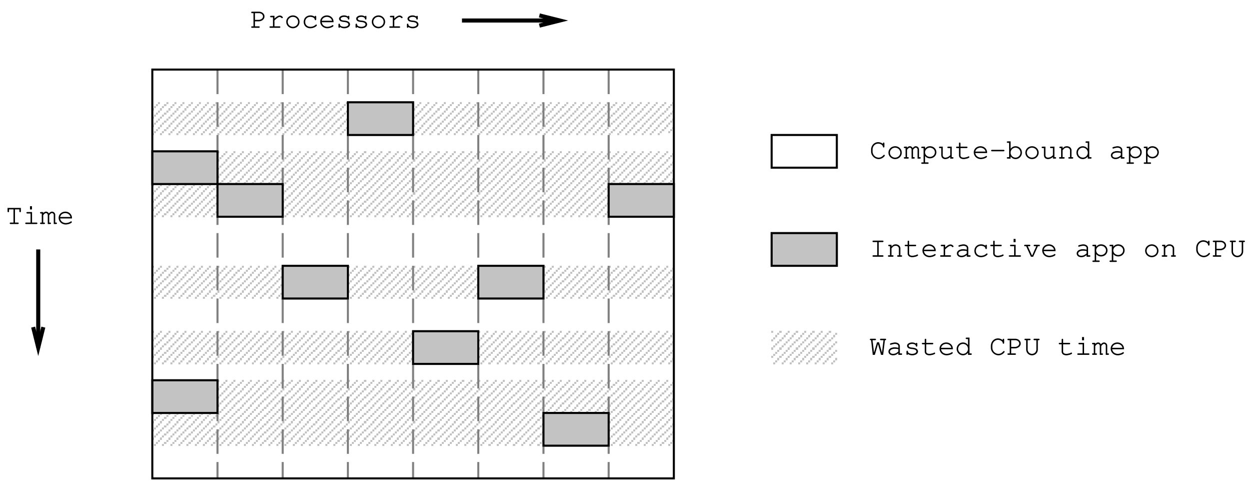


Figure 7-4. A compute-intensive application that also synchronizes intensely may slow down considerably when multiprogrammed with an interactive application. In the worst case, the compute application may only make forward progress when *all* of its processes are scheduled on the machine simultaneously. If demand interactive processing occurs randomly and independently across the processors, much of the CPU time may be wasted.

another, a process may be delayed simply because a peer process has been delayed. In the worst case, if the compute-intensive application synchronizes very frequently, it may require all of its constituent processes to be scheduled near-simultaneously to make progress. Figure 7-4 illustrates the extreme case where much of the CPU time is unusable to the compute-intensive application because of interruptions. The problem of unusable CPU time becomes worse as the number of processors increases and as the amount of CPU time devoted to the interactive application increases.

Assume f represents the fraction of CPU time used by the interactive application. The slowdown of the compute-intensive application ranges from the best case (Figure 7-3) of

$$\frac{1}{(1-f)}$$

to a worst case (Figure 7-4) of

$$\frac{1}{(1-f)^P}$$

for P processors if work arrives independently for each processor. Arpaci *et al* [3] document this effect in a network of workstations for disruptions caused by periodic daemon processes.

Coscheduling improves the performance of the compute application although possibly at the expense of the interactive application. In coscheduling mode, the scheduler allocates fixed-size timeslices across the machine to each application using synchronized clocks. The effect of coscheduling is illustrated in Figure 7-5. Coscheduling wastes CPU time because the interactive application cannot necessarily use all of its assigned time slice. Coscheduling also can dramatically increase the response time of the interactive application. However, coscheduling limits the slowdown observed by the compute-intensive application to a known value. With fixed-size timeslices, the slowdown with two applications is limited to two. A more elaborate coscheduler than ours could adjust the relative size of the timeslices to further improve slowdown and to reduce wasted CPU cycles.

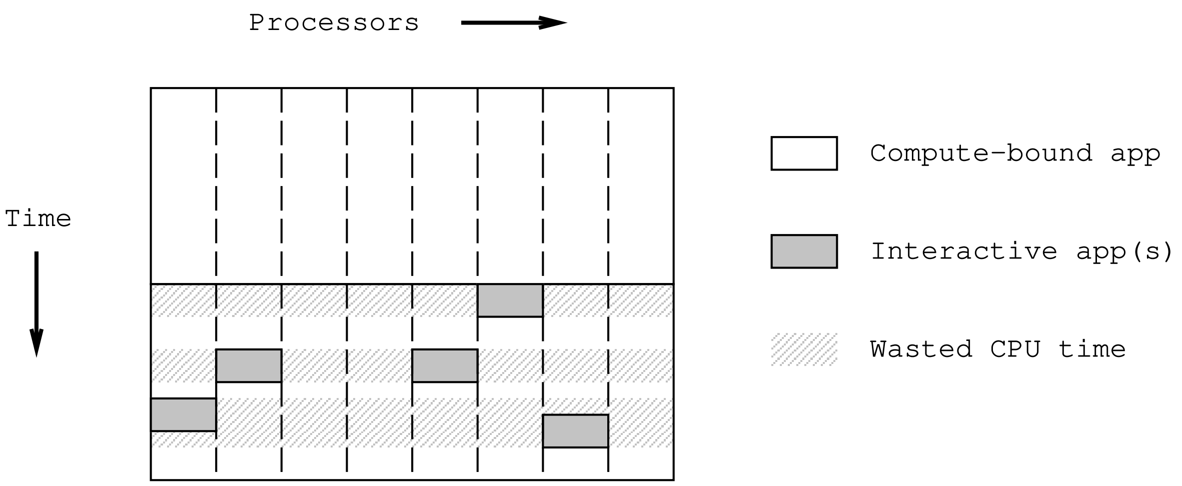


Figure 7-5. Coscheduling assigns blocks of time across multiple processors to a single parallel application. Here, half of the CPU time on all the processors is assigned to the compute-intensive application, so the slowdown of that application is at worst a factor of two. The other half of the CPU time is scheduled in a conventional, independent fashion. The coscheduling pattern repeats periodically in time.

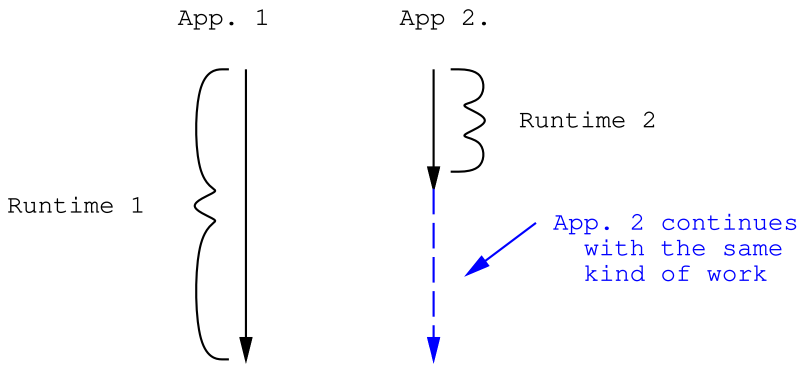


Figure 7-6. Multiprogrammed workloads requires some effort to measure. For our experiments, the faster application is kept running performing the same kind of work so that the measurements are kept symmetric. The runtime measurement thus accurately represents the "instantaneous" effect of one application on another.

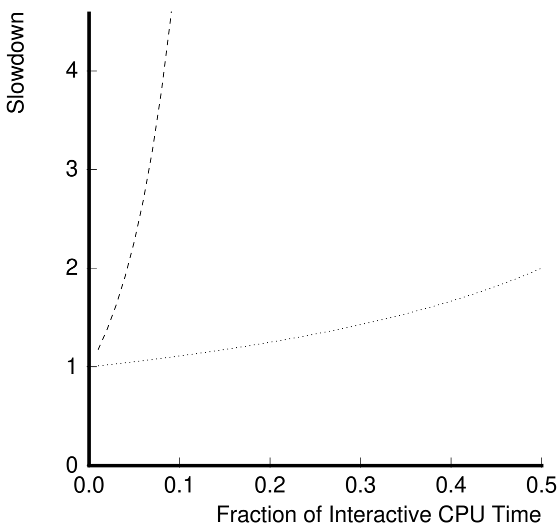


Figure 7-7. The expected range of slowdowns of the compute-intensive application under interactive scheduling varies over a huge range. The slowdown is plotted against the fraction of total CPU time devoted to the interactive application.

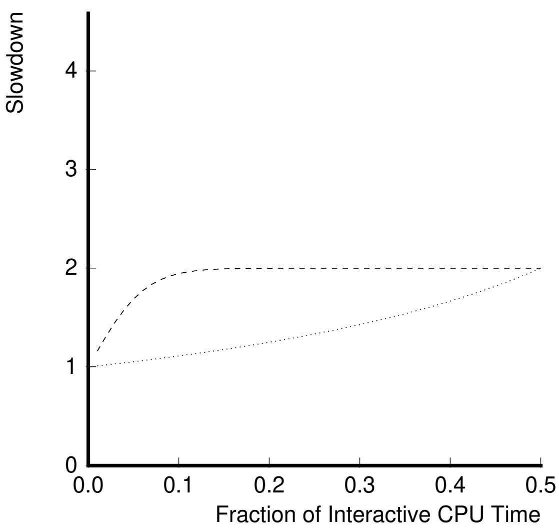


Figure 7-8. The expected slowdown of the compute-intensive application with fixed-timeslice gang scheduling is limited to a factor of two with two applications.

The chief metric for the experiment is the slowdown of the multiprogrammed application over the same application running standalone. To keep the effects of one application on the other clear, we contrive to measure slowdowns that reflect the "instantaneous" effect of one application on the other. To measure this effect, we measure the time for each application to complete a fixed amount of work. However, as shown in Figure 7-6, the application that finishes its work early continues performing the same kind of work until both applications have finished. This contrivance is easily achieved with synthetic applications. For the real applications, we altered the interactive application (the web robot in Table 7-1) to wait for the compute-intensive application to exit.

The interactive applications are fed work as a random process with exponentially-distributed inter-arrival times. For the experiments, we vary the rate of work across a range. The rate of work is converted into a fraction of CPU time, f, by measuring the CPU time consumed by the interactive job at each rate setting. The CPU time consumed is measured indirectly by performing the experiment with the embarrassingly parallel `null` application and measuring the slowdown of `null`.

Figures 7-7 and 7-8 illustrate the range of expected slowdown effects for a parallel, compute-intensive application when scheduled against an interactive application running under priority scheduling and coscheduling, respectively. The graphs plot the slowdown of the compute-intensive application versus the fraction of CPU time used by the interactive application.

With priority scheduling in Figure 7-7, the slowdown varies across an enormous range bounded by the two dotted lines. The lower dotted line predicts the best possible slowdown based on the amount of CPU time used by the interactive application (Figure 7-3). This slowdown corresponds to $1/(1-f)$ for an embarrassingly parallel application. The upper dashed line predicts the worst possible slowdown based on the amount of time the compute-intensive application has control of all

16 processors (Figure 7-4). The upper dashed line is a function of the number of processors in the system ($1/(1-f)^{16}$ in this case) and, as can be seen, can grow to be alarmingly steep.

With fixed-timeslice coscheduling, the worst-case slowdown is limited to a factor of two with two applications. Coscheduling in the style of Figure 7-5 as used in FUGU allows the compute-intensive application to slowdown somewhat less because some of the priority-scheduled CPU time is recoverable. Figure 7-8 again shows two dotted lines corresponding to the worst and best possible slowdowns. An even better coscheduler that could vary the size of the timeslices could limit the slowdown of the compute-intensive application to $1/(1-f)$ in all cases.

The cost of coscheduling is in its effect on the interactive application. The response time of the interactive application may be increased dramatically by the fact that the interactive application is not runnable for whole timeslices of time. We do not pursue this cost in detail in this thesis. Instead, we assume that there is a tradeoff available; *i.e.,* when the compute-intensive application has a slowdown of more than two, the scheduler clearly has a tradeoff that it is choosing to make. A system that only supports coscheduling or only supports interactive scheduling does not have the opportunity to exploit such a tradeoff.

The slowdown observed in FUGU with two-case delivery is a combination of application effect and message-system effect since some messages are diverted from the fast path. Section 7.2.2 presents results for the mixed workload with real applications, illustrates the application effects and makes qualitative points about the effect of two-case message cost. Section 7.3 will examine the two-case tradeoff in detail.

7.2.2 Mixed Workload with Real Applications

The parameterized mixed workload experiment described above serves as a means to distinguish between message system effects. This section presents the results of running several real applications together in the mixed workload.

In our experiment, we run pairs of compute-intensive/interactive applications on a simulated, 16-node system. The compute-intensive application in each case is one of the four real parallel applications listed in Table 7-1. The interactive application consists the web server from Table 7-1 plus one web robot as a client per node. Each robot generates requests at a controlled rate. We vary the rate to vary the amount of CPU time used by the interactive processes and thus the amount of CPU time available to the compute-intensive application.

The slowdowns observed by our sample parallel applications in the mixed workload experiment under interactive scheduling are moderate. Figures 7-9 and 7-11 illustrate the effect of the interactive application on the compute-intensive application when the system scheduler performs independent, priority-based scheduling.[1] Figure 7-9 shows the slowdown of each compute-intensive application multiprogrammed over the same application run standalone versus the fraction, f, of each processor's CPU time devoted to servicing the interactive application. The slowdown is again bounded by dashed lines at $1/(1-f)$ and $1/(1-f)^{16}$. The actual slowdowns can be observed to fall between these extremes, depending on application characteristics. The `enum` application synchronizes infrequently and thus comes nearest to the minimum slowdown of $1/(1-f)$. The other applications synchronize

[1] The slowdown of the interactive application under independent, priority-based scheduling is always close to one.

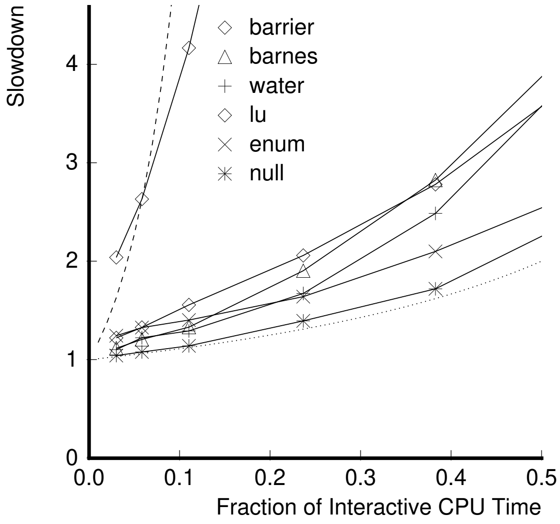


Figure 7-9. The performance of the compute-intensive application in a mixed workload with interactive scheduling is plotted against the fraction of CPU time devoted to interactive tasks (16 processors).

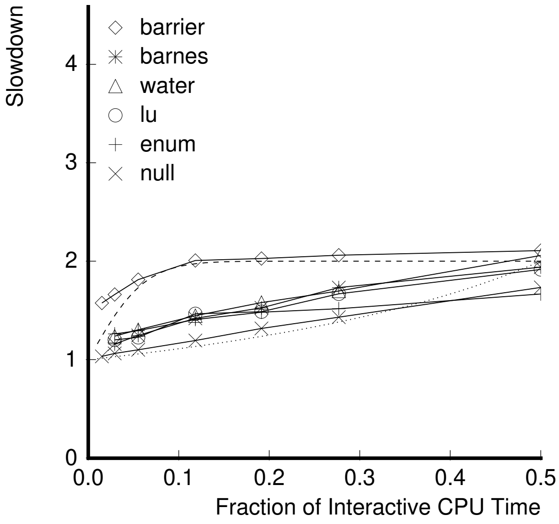


Figure 7-10. The performance of the compute-intensive application in a mixed workload with coscheduling is plotted against the fraction of CPU time devoted to interactive tasks (16 processors).

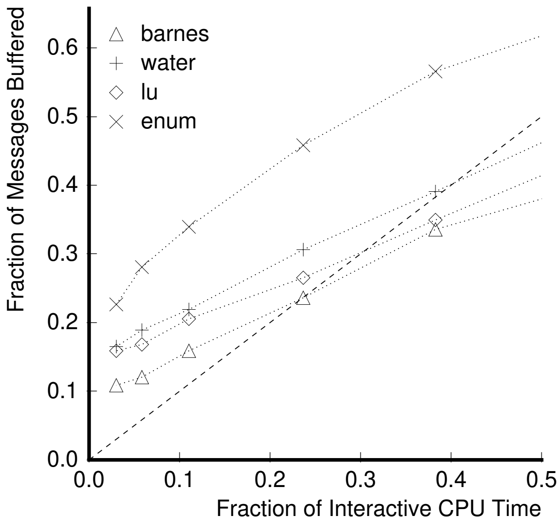


Figure 7-11. The fraction of messages taking the buffered path, f_{buf} in a mixed workload with interactive scheduling versus the fraction of CPU time devoted to interactive tasks (16 processors). The dashed line at $f_{buf} = f$ represents a naive expectation.

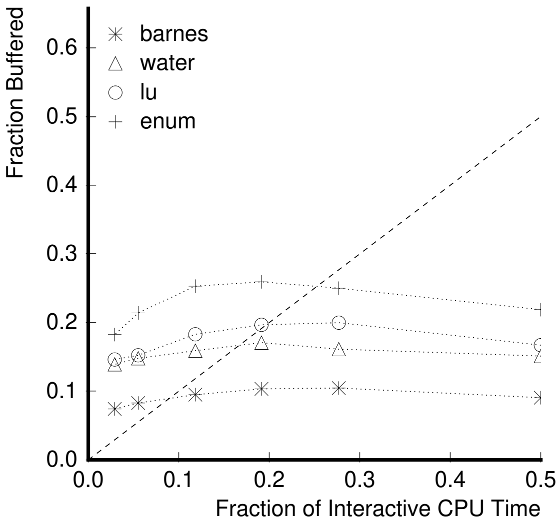


Figure 7-12. The fraction of messages taking the buffered path in a mixed workload with coscheduling versus the fraction of CPU time devoted to interactive tasks (16 processors).

a moderate amount but show nowhere near the slowdown of `barrier` in Figure 7-7.

The fraction of messages buffered by the sample applications in the experiment roughly follows the fraction of interactive CPU time, as one would expect. Figure 7-11 shows the fraction of total messages, f_{buf}, in each compute-intensive application that take the buffered path versus the fraction of CPU time devoted to the interactive application. The fraction f_{buf} is nonzero as f approaches zero because of effects other that the interactive application. In particular, the microkernel-style system scheduler is another user-level application that runs periodically and thus induces a fixed amount of buffering.

The fraction of messages buffered rises with a slope less than unity, eventually falling below the $f_{buf} = f$ line for the CRL applications, because synchronization-induced slowdown tends reduce the demand for buffering. In an application with little synchronization, like `enum`, the number of messages buffered tends to follow f. If one process in the application is stalled and buffers a message due to multiprogramming, the other processes in the application continue sending messages at the same rate. However, in an application that synchronizes, the number of messages buffered is lower. If one process in the application is stalled and has to buffer a message, some other process may slow down because it is waiting on a reply to *that* message. The result is that the demand for buffering falls.

The slowdown effects observed by parallel applications other than `barrier` under interactive scheduling are moderate. For comparison, Figures 7-10 and 7-12 plot the slowdown and fraction of messages buffered, respectively, for the real applications under fixed-timeslice coscheduling as described in conjunction with Figure 7-5. The slowdowns are limited to a factor of about two, as expected.[2] The fractions of messages buffered are similarly limited.

There are two points to take away from the plots, both qualitative. The first is the main point of the thesis: two-case delivery is justified in much of the graph in Figure 7-9. The performance of each compute-intensive application smoothly approaches a (perfect) slowdown of one at the left as the CPU time used for interactive work diminishes. Optimistic message delivery is justified in this regime because most messages are delivered to the correct process immediately and few are buffered. From Figure 7-11, when about 10% of the CPU time is devoted to interactive tasks, only about 45% of messages are taking the buffering path. In this regime, an alternate system with hardware-managed buffering pays a performance and implementation cost penalty for hardware that is going unused.

Second, towards the right of Figure 7-9, the slowdowns observed eventually justify coordinated scheduling of jobs instead of independent scheduling of processes. Some of the applications in Figure 7-9 exhibit slowdowns disproportionate to the amount of CPU usage, as explained in conjunction with Figure 7-4. The slowdown is due to application characteristics, not the message system, because there still are not very many messages buffered (Figure 7-11). With FUGU, the decision to switch to coscheduling becomes a tradeoff of message cost versus scheduling flexibility. A system such as the CM-5 that uses coscheduling for protection always suffers the slowdown of coscheduling.

> *Conclusion 2: The direct virtual network interface is desirable because most messages are delivered via the fast path.*

[2]Some of the slowdowns are still less than two at the extreme right due to a scheduler bug. Similarly, the fraction of messages buffered is non-zero at the right side of the graph due to the same bug.

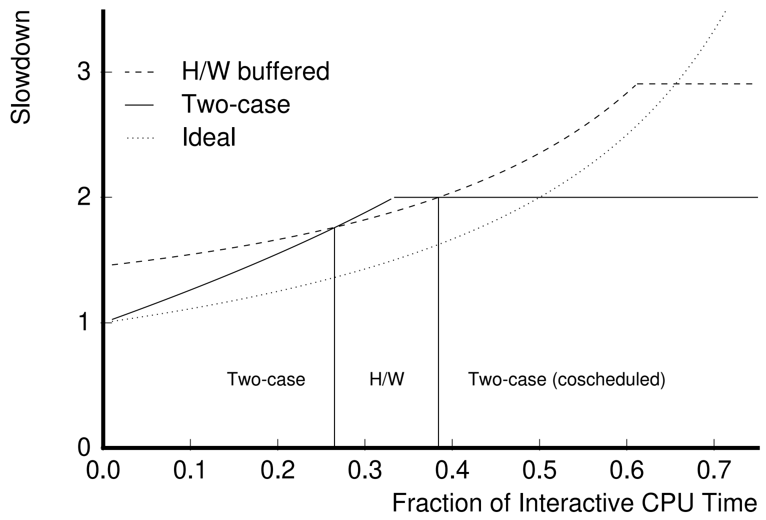


Figure 7-13. An artist's conception of a comparison of hardware buffering to two-case delivery. Two-case delivery is desirable at the left because relatively few messages are buffered. Two-case delivery is desirable at the right if coscheduling is used and application characteristics may demand coscheduling anyway.

In short, at the far left, two-case delivery is desirable because most messages are delivered via the fast path, even with independent, priority-based scheduling favoring the interactive application. At the far right, two-case delivery is desirable if the system switches to coscheduling and it is probably useful to switch to coscheduling for application reasons alone. The tradeoff in the middle of the graph is less clear. The next section explores this tradeoff.

7.3 Mixed Workload Analysis

This section compares the performance of the direct VNI approach with two-case delivery and software buffering to a hypothetical system using hardware-supported buffering in main memory. The comparison uses experimental measurements from Section 7.2.2, an analytical model, parameters from the direct VNI implementation in FUGU and estimated parameters for possible implementations of a hardware-supported memory-based approach. The results are favorable to the direct VNI with one exception. Applications that are embarrassingly parallel do not need to be coscheduled. An application that is both communication-intensive with small messages and embarrassingly parallel application can perform better with hardware buffering than with the direct VNI unless switched to coscheduling. However, such applications are amenable to many other techniques as well, such as explicit batching through a library that performs source buffering. Finally, as an extra result, we show analytically that the advantages of coscheduling increase as communication overhead and latency drop. In other words, if coscheduling is not widely recognized as a problem now, it may be because network interface overheads are generally high.

The basic tradeoff is between the cost of messages handled immediately and the cost of messages

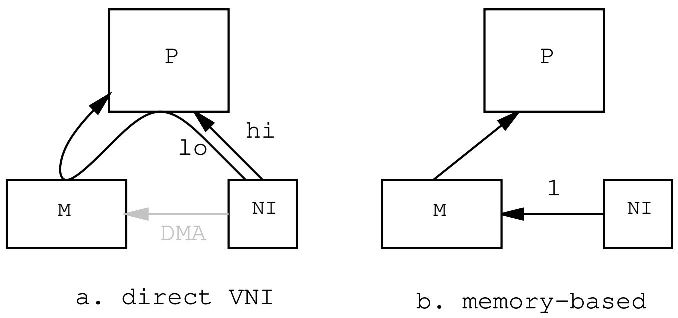


Figure 7-14. Using the direct VNI, (a), messages are received by the application with high probability or take a software-managed route through memory with low probability. In a memory-based system, (b), all messages traverse memory although the NI-to-memory connection is assumed to have hardware support.

handled after some delay. With two-case delivery, messages handled immediately cost the least because they take the fast, direct path. Messages that are delayed cost more because of the extra overhead of buffering. With hardware-supported buffering, the opposite is true, messages handled immediately suffer extra cost over direct delivery because of the latency of the trip through memory. Messages that are delayed, however, may be handled quite efficiently if they can be handled in batches.

Figure 7-13 sums up the comparison in terms of the mixed workload experiment from the previous section. The ideal curve represents application slowdown if a direct delivery path could be used for every message. The ideal is naturally unattainable. The two-case delivery curve adds a cost that increases with f, the fraction of CPU time used by the interactive application. The hardware-buffered curve adds a cost that decreases with f. For low f, two-case delivery has a clear advantage. For high f, hardware buffering has the advantage. However, for all but embarrassingly parallel applications, it is advantageous (at least to the compute-intensive application) to switch to a coscheduling strategy once the slowdown exceeds two. With coscheduling, two-case delivery will again do well. This section quantifies the range of tradeoffs using measurements of applications running on FUGU combined with a simple model to predict performance with with hardware-supported buffering.

Model. We propose a simple model for the runtime of the compute-intensive application in the mixed workload experiment for the purpose of separating out the effects due solely to the message system. We then use the model to predict performance with a different message system that uses hardware-supported buffering in memory. The model is simplistic, but the assumptions made are conservative given the intended purpose of predicting performance under hardware buffering. The runtime is taken to be the sum of three terms:

- R_0 is the standalone execution time without message overhead.
- $R_{other}(f)$ is additional runtime due to the application-specific effects of the interactive application when the interactive application consumes a fraction, f, of the available CPU time.
- A third term representing message overhead in terms of O_{dir} and O_{buf}, the overhead of handling a message immediately and after buffering, respectively, f_{buf}, the fraction of messages

that must be buffered and M, the average number of messages per processor.

The runtime of an application in the mixed workload experiment, R_{mix}, is then:

$$\text{mixed runtime, } R_{mix} = R_0 + R_{other}(f) + M \cdot ((1 - f_{buf}) \cdot O_{dir} + f_{buf} \cdot O_{buf}) \tag{7.1}$$

In contrast, the standalone runtime, R_{sa}, can be computed by assuming that the time due to multiprogramming, R_{other} is zero and that the fraction of messages buffered, f_{buf} is zero:

$$\text{standalone runtime, } R_{sa} = R_0 + M \cdot O_{dir} \tag{7.2}$$

Finally, the runtime under coscheduling with fixed-size timeslices, R_{co}, is taken to be at worse two times the standalone runtime:

$$\text{coscheduling runtime, } R_{co} = 2 \cdot R_{sa} = 2\,(R_0 + M \cdot O_{dir}) \tag{7.3}$$

The model terms may be interpreted in terms of the four plots in Section 7.2.2. The y axis in Figure 7-9 is R_{mix}/R_{sa}: the slowdown in the plot is the ratio of the runtime under the mixed workload to the runtime standalone. Figure 7-11 plots f_{buf} against f for the same experiment. The y axis in Figure 7-10 is the slowdown under coscheduling. The slowdown under coscheduling, R_{co}/R_{sa}, is taken to be a constant two as a worst case in the model. Figure 7-10 shows actual slowdowns of less than two for all the applications except the synthetic `barrier`, but Figure 7-10 represents a more relaxed form of coscheduling than fixed timeslices. The scheduler used to produce Figure 7-10 mixes fixed timeslices with priority-scheduled timeslices (as depicted in Figure 7-5), allowing the compute-intensive job to scavenge extra unused CPU time. Accordingly, the fraction of buffering, f_{buf} for coscheduling plotted in Figure 7-12 is not zero as Equation 7.3 suggests because some amount of priority-based scheduling is still going on.[3]

Given the model and measurements of the parameters for the direct VNI system, we can proceed to predict the performance of the same applications with a different message system. We discuss the parameters of the direct VNI in FUGU and the estimated parameters for a hardware-supported buffering system next, and then come back to analyzing the tradeoff between two-case delivery and hardware buffering depicted in Figure 7-14.

Numbers. Table 7-4 gives measured and estimated possible values for the O_{dir} and O_{buf} parameters in the mixed workload model using interrupt-based delivery. There are three sets of numbers. The first set consists of the values measured from our direct VNI implementation (Tables 4-4 and 4-5) along with estimates for a range of costs with a hardware buffered system. The hardware buffering O_{dir} cost is estimated to be the direct-path upcall cost in FUGU, plus 42 cycles to manipulate the queue (based on the queue extract cost in Table 4-5 minus a 10-cycle cache miss time), plus a cache miss time. Two numbers are given to represent a range of cache fill times from memory. The lower bound corresponds to 10 cycles, as in the FUGU prototype hardware. Cache refill times in more modern machines have climbed far beyond this number, however; 200 cycles is used as an upper bound. The O_{buf} time for hardware-supported buffering is estimated to be the just queue extract

[3]The fraction f_{buf} should be zero at the left and right extremes of Figure 7-12. It is nonzero on the left because of the effects of fact that the scheduler itself is another application that exerts a constant effect on the compute-intensive application. The fraction is nonzero at the right due to the scheduler bug.

Hardware	Mode	O_{dir} cycles	O_{dir} rel.	O_{buf} cycles	O_{buf} rel.
Current implementation	Buffer on mismatch *(measured)*	115	(1)	232	(2.02)
	Hardware buffering *(estimated)*	167- 357	(1.45) (3.10)	52 52	(0.45) (0.45)
Aggressive reimplementation *(fast upcall, in-kernel enqueue)*	Buffer on mismatch *(estimated)*	87	(1)	80	(0.92)
	Hardware buffering *(estimated)*	139- 329	(1.60) (3.78)	52 52	(0.57) (0.57)
Miscellaneous for comparison *(see text)*	Upcall always *(measured)*	115	(1)	145	(1.20)
	Buffer always *(measured)*	1400	(12)	232	(2.02)

Table 7-4. Measured and estimated parameter values for the model of the mixed workload experiment. Values are given both in cycles and in a relative value normalized to the O_{dir} for the two-case delivery implementation in each group. Estimates for hardware buffering are given as a pair based representing a range of main memory fill times of 10 cycles (in the actual FUGU hardware) to 200 cycles.

cost of 52 cycles from Table 4-5. We assume, optimistically to both hardware- and software-based buffering, that any cache miss cost in the buffered case can be successfully amortized or tolerated over a batch of buffered messages.

We will use the measured FUGU numbers from the first set of numbers in Table 7-4 in the comparison below. In retrospect, though, the direct VNI implementation in FUGU performs only moderately well. The overhead of buffer enqueue in software is a particular limit as we will see shortly below and then again in Section 7.4.2. The second set of numbers in Table 7-4, presented here for comparison, are speculative estimates of overhead costs in a re-implementation of the direct VNI. The protected upcall in the direct case could be reduced from 115 cycles to 87 cycles using atomicity hardware, as shown in Table 4-4. Further, the common case of the queue insert interrupt handler could be speeded up significantly. The current implementation always uses an upcall to the library part of the Exokernel to insert a message into the software queue with a common-case cost of 180 cycles. This cost can be reduced by tail-polling or by installing the common case insert handler into the kernel. Based on an in-kernel implementation, we speculated that the overhead could be as low as 30 cycles.

The description and analysis of the two-case interface has been based on a policy of buffering all messages that arrive with a mismatched GID. The FUGU implementation is a bit more flexible than that and, although we do not pursue the alternatives further in this thesis, the third set of numbers in Table 7-4 present measurements for two alterate policies. First, buffering on mismatches is only a policy; the operating system in fact supports message delivery via a cross-domain upcall in mismatched-GID situation. Cross-domain upcalls cost only 30 cycles more than intra-domain upcalls, so this mode is quite efficient. The tradeoff between upcall-always and buffer-on-mismatch mode is an interesting question for further research. Second, it is also possible to configure the operating system to buffer *all* messages. The overhead of delivering a single, isolated message (O_{dir}) with buffering is very high, about 1400 cycles, in the implementation due to the cost of thread scheduling.[4] This 1400 cycle overhead is extremely high for the domain of small message delivery in a scalable workstation but is roughly the cost of an interrupt in some current workstation operating systems.

Tradeoff in Real Applications. Using the model, measurements and estimates above, we can construct the tradeoff between two-case delivery and hardware buffering posed in Figure 7-14 for the data from the real applications in the mixed workload experiment.

We assume that the execution times used to produce Figure 7-9 in Section 7.2.2 correspond to R_{mix} for two-case delivery (call it R_{mix_2c}) in Equation 7.1. The rest of the terms except $(R_0 + R_{other}(f))$ are known, so we can reconstruct an estimate of the runtime under hardware buffering, R_{mix_hw}, by solving for $(R_0 + R_{other}(f))$ and substituting the value into an equation for R_{mix_hw}. In equations, assume:

mixed two-case runtime, $R_{mix\_2c} = R_0 + R_{other}(f) + M \cdot ((1 - f_{buf}) \cdot O_{dir\_2c} + f_{buf} \cdot O_{buf\_2c})$ (7.4)

and:

mixed two-case runtime, $R_{mix\_hw} = R_0 + R_{other}(f) + M \cdot ((1 - f_{buf}) \cdot O_{dir\_hw} + f_{buf} \cdot O_{buf\_hw})$ (7.5)

where f_{buf} comes from the data used to produce Figure 7-11, M is taken to be the average number of messages per processor as quoted in Table 7-2 and the overhead cycle counts come from Table 7-4.

Equations 7.4 and 7.5 embody a number of assumptions by using the same values for $(R_0 + R_{other}(f))$, M and f_{buf} in the two cases. However, given that R_{mix_hw} exceeds R_{mix_2c} (as we shall see), the assumptions conservatively favor the hardware case:

- The CPU and application slowdown effects, $R_{other}(f)$, will stay the same or go up if message overhead goes up; assuming $R_{other}(f)$ stays the same favors the system with higher costs.
- The fraction of messages delayed, f_{buf}, may go *down* with hardware buffering since hardware buffering clears the buffer more quickly.
- The number of messages, M, is not likely to change.

Figures 7-15 and 7-16 plot measured two-case slowdowns and predicted hardware slowdowns for the barnes and `enum` applications, respectively, against the fraction of CPU time devoted to interactive work, f. These two applications were chosen because they exhibit the highest messaging rates for the real applications. Each graph includes four curves. All the curves represent execution times normalized to R_{sa_2c} (R_{sa} with two-case delivery, *i.e.,* standalone runtime as measured) for each application. The bottommost of the four curves is R_{mix_2c} (measured) up to the point where it equals R_{co_2c}, at which point we assume the user either switches to coscheduling or chooses not to as part of a deliberate performance tradeoff. The next two curves (shaded in between) bound a range

[4] The buffer-always policy invokes the lower-right sequence of operations in Table 3-1 for every message. The scheduler manipulations along this path are not particularly optimized in FUGUbecause they have little impact in other, normal modes of operation.

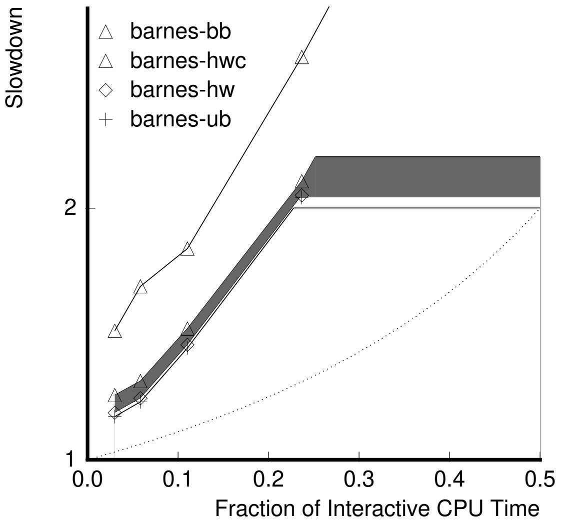


Figure 7-15. Performance tradeoff for the `barnes` application (16 processors).

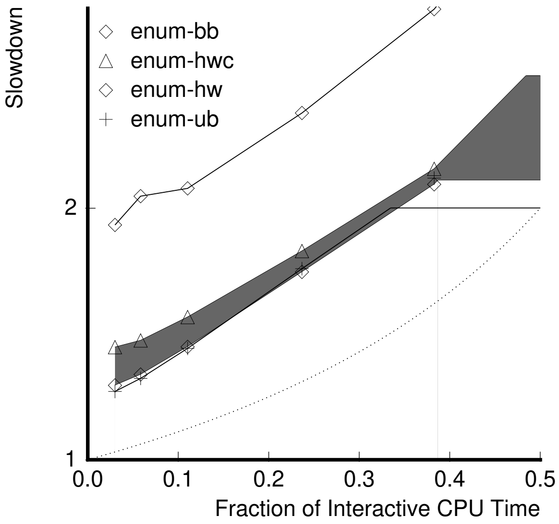


Figure 7-16. Performance tradeoff for the `enum` application (16 processors).

of (normalized) R_{mix_hw} execution times computed by using the values for O_{dir} and O_{buf} given in the top section of Table 7-4. The R_{mix_hw} curves are also flattened off at the point where they equal R_{co_hw}. Finally, for a gross comparison, the topmost line in each of the two plots is the normalized time measured using the alternate, buffer-always policy which incurs 1000+ cycle thread switches instead of upcalls in the direct case.

The interpretation of Figures 7-15 and 7-16 is that window in which hardware-supported buffering in memory outperforms two-case delivery is very small. Two-case delivery performs well at the extremes, as noted in the previous section. In the middle ranges of f, hardware buffering outperforms two-case delivery in `enum` for very small (*i.e.,* improbably small) cache miss costs. As mentioned earlier in connection with Table 7-4, the two-case delivery performance could be improved quite a bit which would make the comparison with hardware buffering even more favorable. Thus we refine the last conclusion:

> *Conclusion 3: The fact that most message are delivered via the fast path justifies not only two-case delivery, but also the use of software in the buffered path.*

Physical Interpretation of Equations. The "breakeven" point between two-case delivery and hardware buffering leads to an insightful physical interpretation. We can compute the point at which two-case delivery is no longer attractive compared to hardware-supported buffering by setting Equations 7.4 and 7.5 equal to one another and solving for f_{buf}:

$$(1 - f_{buf}) \cdot O_{dir\_2c} + f_{buf} \cdot O_{buf\_2c} = (1 - f_{buf}) \cdot O_{dir\_hw} + f_{buf} \cdot O_{buf\_hw} \tag{7.6}$$

$$f_{buf\_breakeven} = \frac{(O_{dir\_hw} - O_{dir\_2c})}{(O_{dir\_hw} - O_{dir\_2c}) + (O_{buf\_2c} - O_{buf\_hw})} \tag{7.7}$$

The terms in Equation 7.7 have a physical interpretation that gives insight into the performance tradeoff. First, $(O_{dir\_hw} - O_{dir\_2c})$ is the time for buffer management and buffer cache misses incurred in hardware buffering but not in a direct interface. Second, $(O_{buf\_2c} - O_{buf\_hw})$ is exactly the time of the buffer enqueue handler used for software buffering.

$$f_{buf\_breakeven} = \frac{(O_{\text{buffer management}} + O_{\text{cache fill}})}{(O_{\text{buffer management}} + O_{\text{cache fill}}) + O_{\text{software enqueue}}} \tag{7.8}$$

Equation 7.8 (Equation 7.7 rewritten) makes it clear what is being traded. The extra costs of buffered over direct gives us something to trade. The software enqueue handler is what is being traded. We can make the tradeoff at all because we expect to coschedule when f (and therefore f_{buf}) exceeds some threshold like two. The main point made by Equation 7.8 is that the software enqueue handler needs to be fast. In particular, we need to avoid cache misses in that handler. It is possible to do so because our DMA engine forwards data from from network to the DRAM without going through the caches or processor.

Importance of Coscheduling. Finally, the there is an extra point to extract from the tradeoff between the direct VNI and a hardware buffered system: coscheduling is more important when message overheads are lower. Message overhead adds a constant cost to the runtime of the applications under interactive scheduling. But it adds a multiplicative slowdown with coscheduling. The result is that coscheduling increases in importance as message overhead is reduced. The implications of the equation can be observed graphically in Figures 7-15 and 7-16.

Two-case delivery under coscheduling has a slowdown of $2\times$. The two-case delivery curve under interactive scheduling observes a $2\times$ slowdown for a pretty small f, so there is an early incentive to switch to coscheduling. Hardware buffering under gang scheduling suffers a slowdown of more than $2\times$ because of the extra overhead involved. The hardware buffering curve under interactive scheduling reaches the coscheduling line at a point that is consistently to the right (higher f) than the switchover point for two-case delivery. This observation helps mitigate the fact that two-case delivery occasionally demands help from the system scheduler: improving the performance of *any* network interface has the effect of increasing the reliance on the system scheduler.

The mixed workload experiment described above demonstrates the flexible protection enabled by the direct VNI approach: we are able to run multiple applications together using scheduling policies as the applications themselves demand. The mixed workload experiment let us draw two conclusions about the performance of two-case delivery. First, two-case delivery offers a clear advantage when either the rate of interactive disruptions is low or disruptions are sufficient to prompt a switch to coscheduling. The advantage comes because under these conditions, most messages take the direct path, giving the direct VNI performance close to that of unprotected hardware. Secondly, the software overheads achievable in a implementation of two-case delivery are low enough to allow two-case delivery to outperform a system using hardware-supported buffering in memory. Using hardware to support buffering in memory amounts to supporting an uncommon case. The tradeoff only looks more favorable to two-case delivery with technology trends and aggressive applications. The next section moves on to the other major question in the direct VNI: how to manage the potentially unbounded demand for buffering.

7.4 Buffer Consumption

The virtual buffering system in the direct VNI supports performance and programmability by providing the illusion of unbounded buffering. Maintaining this illusion would be costly or impossible if applications tend to abuse it. We approach the problem of unbounded demand for buffering from two perspectives in this section and the next section. First, in this section, we show that "well-behaved" applications naturally avoid extensive buffering. Our sample applications are found to be well-behaved and we define the limits on acceptable application behavior. Second, section 7.5 evaluates one technique, overflow control, for applying feedback to an ill-behaved application by way of scheduling mechanisms. Overflow control effectively reduces the demand for buffering or, at worst, turns buffer consumption into a slowdown that affects the offending application alone.

This section details experiments that induce buffering artificially in applications using the direct VNI. The results make two points. First, ordinary applications naturally avoid buffering mode. Physical memory requirements tend to be low even under adverse scheduling conditions. Virtual buffering thus improves memory performance because it avoids consuming physical memory unless the application requires it. Second, experiments with a synthetic application show in detail the limits on application behavior required to avoid buffering. The conclusion is that it is still important to keep a low overhead in the buffered case because the batch throughput of the buffered case sets a limit for certain programming styles that synchronize infrequently.

There are two experiments. One induces buffering in real applications though an artificial policy and artificial scheduling. The second is a synthetic application study that attempts to define the region of "stability" with respect to buffering. The stability is applicable to any system with a direct interface, but particularly interesting in Fugu where we promise effectively unbounded buffering.

7.4.1 Artificially Induced Buffering

We evaluate the performance of our implementation by examining the effects of buffering in the applications described in Section 7.1 under conditions that induce buffering artificially. The conditions are as follows. First, we use a buffer-on-mismatch policy so that buffering can be conveniently induced by the scheduling effects alone. Second, we modify our gang scheduler to deliberately introduce "skew" between the scheduling times of applications on different processors, as depicted in Figure 7-17. This (perverse) scheduling arrangement allows us to generate arbitrarily bad scheduling in a controlled manner.

The experiment is to run each application multiprogrammed with a "null" (busy-waiting) application with varying amounts of scheduling skew. Skew will induce buffering through the buffer-on-mismatch policy. The scheduler gang-schedules the pair of applications using the local cycle count register on each node as a cue to perform a gang switch. The schedule quality is varied by skewing the cycle count register on each node to produce artificially poor schedules in a controlled manner. This skew creates a window at the beginning and end of each timeslice during which arriving messages will generate a *mismatch-available* interrupt, forcing the application into buffered mode. The skew is varied from zero (perfect gang scheduling) to 90% across the machine. We measure the effects on the real application in each case.

The runtime represents either the third iteration for the iterative applications (`water` and `barnes`) or the whole program. The runtime represents all the cycles used on behalf of the

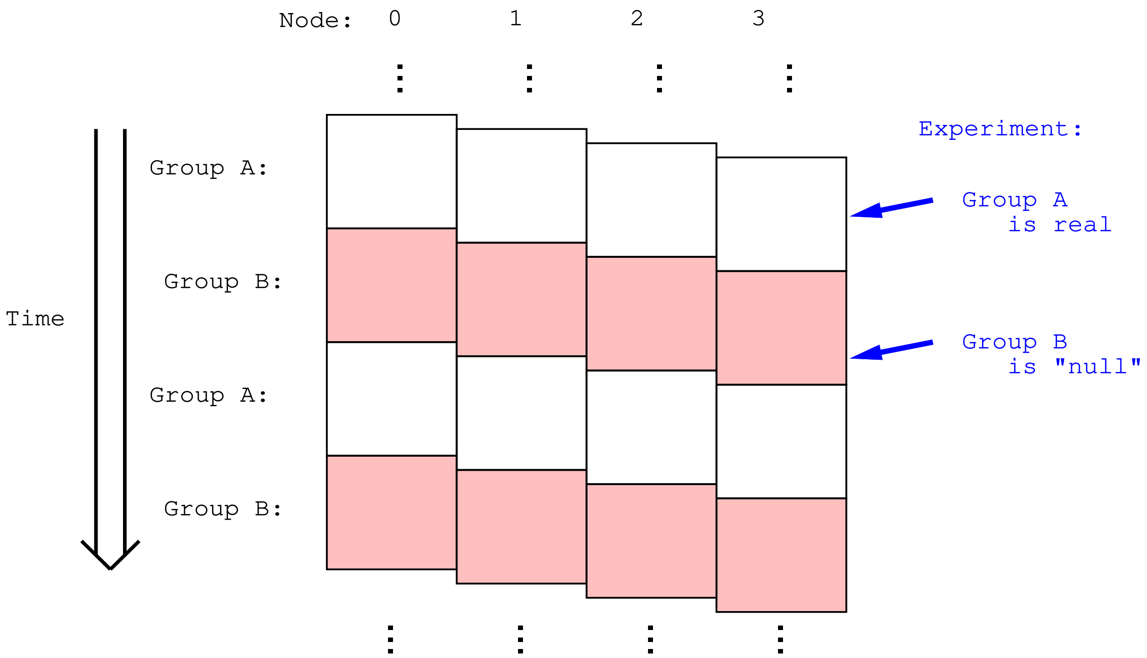


Figure 7-17. Buffering is induced by using coscheduling with "skew". Here, a segment of a timeline for process scheduling on a four-processor machine is shown. The application under test, (A), is multiprogrammed with a synthetic null application, (B) that does nothing but busy-wait. In addition, the timeslice clock is skewed uniformly from processor to processor so that processes are deliberately mis-scheduled for an interval around the beginning of each timeslice.

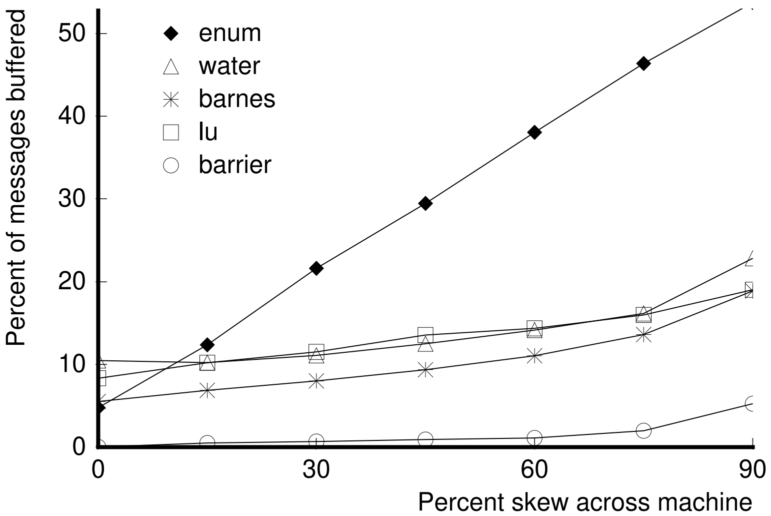


Figure 7-18. The fraction of messages buffered for applications multiprogrammed with a null application is plotted against decreasing schedule quality (eight processors). The fraction is limited by synchronization effects in the CRL applications.

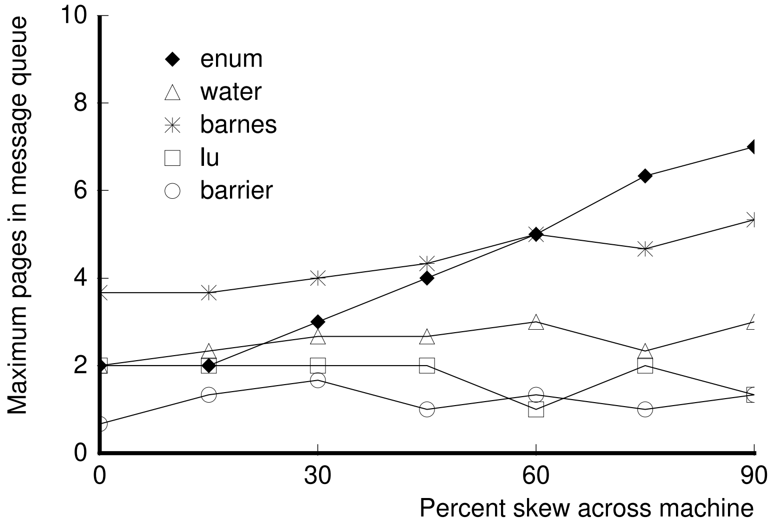


Figure 7-19. The maximum pages of virtual buffer space required per processor for applications multiprogrammed with a null application remains low across the range of scheduling quality (eight processors).

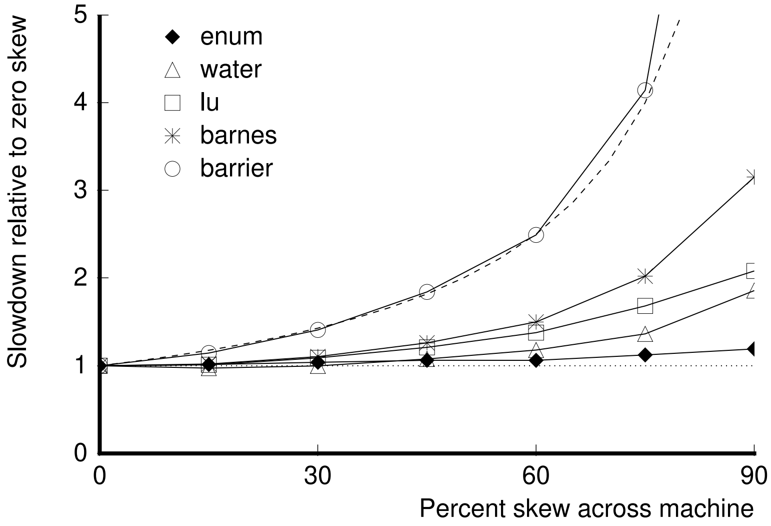


Figure 7-20. Relative runtimes of applications multiprogrammed with a null application are plotted against decreasing schedule quality (eight processors). Runtimes are normalized to the runtime with perfect gang scheduling, which is within 1% of 2× the runtime of the application running alone.

application, including the cost of buffer insertion handlers that actually run while “null” is scheduled. We use a null application rather than two copies of a real application because the experiment is more easily controlled.

Figure 7-18 makes the main point of the experiment: that the demand for buffering is relatively small and increases gracefully. Figure 7-18 plots the fraction of messages that take the buffered path versus decreasing scheduler quality. The applications with intrinsic synchronization exhibit essentially a constant fraction of messages buffered corresponding to the maximum number of messages that can be outstanding simultaneously in the application. `Enum` exhibits buffering linearly with skew as expected for an application with many messages and little synchronization: the likelihood of a message arriving when a process is not scheduled is proportional to the skew between processors.

The maximum number of physical pages required during any run is low, less than seven pages/node, in all cases. The total is small in each case either because the number of messages outstanding is limited or because (in the case of `enum`) the messages are small and are accumulated at only a moderate rate compared to the length of a timeslice. Because the required buffer space is small in the common case, the virtual buffering system will only rarely need to page to disk or invoke the overflow control system.

The applications in the experiment slow down with increased skew largely because of the skew itself and to a small extent because of the cost of buffering. Figure 7-20 lists the relative runtime of each application normalized to the runtime of the application run with zero skew, which is within 1% of 2× the runtime standalone. The `barrier` application is very sensitive to skew because it makes progress only when all processes in the job are simultaneously scheduled: its slowdown is almost exactly the inverse of the skew. Because the `enum` application tolerates latency well it is relatively insensitive to poor schedule quality. The runtime increase in `enum` is due only to the added cost of message buffering. Although the Barnes, Water and LU applications are sensitive to latency, they communicate less frequently than `barrier` and `enum` and so observe intermediate slowdowns.

> *Conclusion 4: Application characteristics can naturally limit the demand for buffering without the introduction of explicit flow control.*

We conclude that the demand for buffering remains low in our applications despite the use of unacknowledged messages and despite (artificially) adverse conditions. In general, we expect applications to suffer buffering overhead only rarely because buffered mode is entered only under unusual conditions and because ordinary applications will clear buffered messages quickly. The second of these expectations is not immediately obvious, so we explore the incidence of buffering with a synthetic application, below.

7.4.2 Limits to Buffering Behavior

If a node must start buffering, several factors help guarantee that the buffer will clear relatively quickly.[5] The first is that any application that requires a reply after message send inherently limits its own communication rate. Limiting the number of outstanding requests guarantees that, if buffering

[5] This section is largely the work of Matt Frank as part of [51]

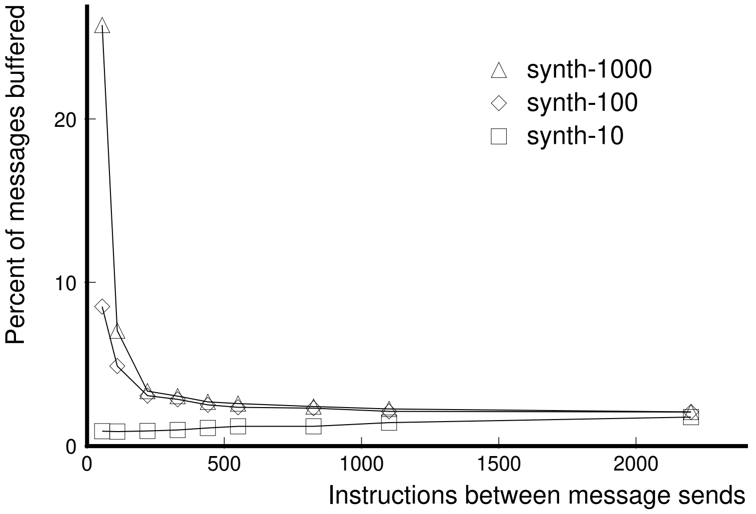


Figure 7-21. Fraction of messages buffered versus send interval ($T_{\text{interhandler}}$) with N messages (for synth-N) sent per synchronization point and 1% scheduler skew (four processors). The fraction is measured over the whole run of the application. The synth-10 application synchronizes sufficiently often that very few messages are buffered. The other applications incur buffering, but only for extremely small values of $T_{\text{interhandler}}$.

mode is entered at all, the maximum number of messages buffered will be finite (and usually small). In addition, the low-level network flow control mechanism guarantees that the maximum short-term injection rate will be limited by the maximum rate at which messages can be diverted (one message per 163 cycles). There remains a class of applications that pass messages and perform little or no synchronization. If such an application launches messages at very high rates for long periods of time, a large fraction of its messages may be diverted through the buffered path. At some point, an application written this way must simply be considered poorly behaved because it will perform poorly on any machine. With the direct VNI, such applications will not interfere with other programs, because the divert mechanism clears messages out of the network quickly, but they will tend to observe both higher average latencies and overheads for message handlers.

Our synthetic application, synth-N, performs producer-consumer communication between four processors with various amounts of synchronization. At the consumer node, each incoming message from the producer invokes a request handler that stalls for a short period, and then sends a reply message. The time to process one of these request messages is fixed in our experiment at 290 cycles, including interrupt and kernel overhead. Reply messages cost 110 cycles. Each node iteratively generates groups of N messages, directed randomly to the other nodes, and then waits for all the acknowledgements from that group of requests, effectively creating a synchronization point and limiting the maximum number of outstanding requests to N. The interval between individual message sends is a uniformly distributed random variable with an average of $T_{\text{interhandler}}$ instructions.

We tested three cases of synth-N with N set to 10, 100 and 1000 messages. The scheduler skew for this experiment was held constant at a small value, 1%, that is sufficient to force the application to enter buffering mode periodically. Figure 7-21 presents the results, giving the fraction

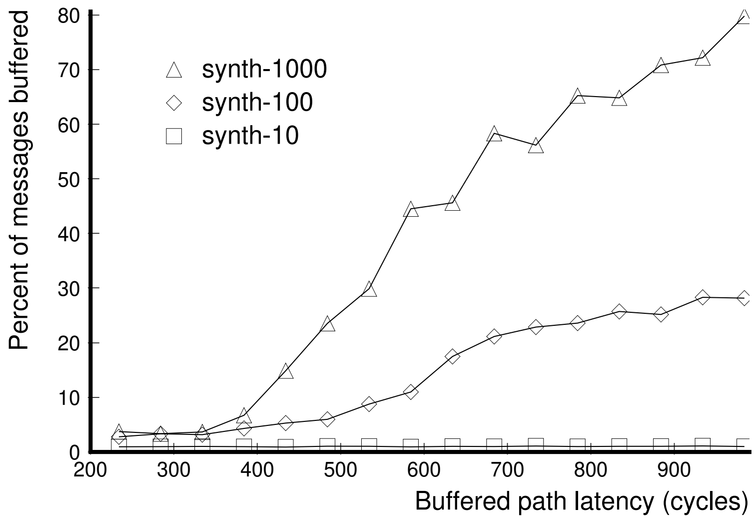


Figure 7-22. Fraction of messages buffered versus cost of the buffered path given $T_{\text{interhandler}}$=275 cycles (four processors).

of messages buffered on the consumer node versus $T_{\text{interhandler}}$. There are two features to observe in the results. First, all versions of synth-N show a small fraction of messages buffered when $T_{\text{interhandler}} > (T_{\text{handler}} + T_{\text{buffering\_reception\_overhead}})$. In this region, the application is well-behaved by virtue of having a low enough send rate so that the consumer's buffer is guaranteed to eventually drain. Second, buffering is reduced as the frequency of synchronization increases (smaller N). In this application, synchronizing has the effect of "manually" clearing the software buffer, so the node is in buffering mode only from the time buffering mode is triggered until the next synchronization. The synchronization in synth-100 and synth-10 occurs more often than timeslices, so these versions are subject to buffering proportionately less often.

On the other hand, Figure 7-22 demonstrates the importance of keeping the cost of the buffered path relatively small. In this experiment $T_{\text{interhandler}}$ is held constant at 275 instructions, but we artificially added latency to the buffer handler. Again, synth-10 buffers only a small fraction of its messages because the benchmark's internal synchronization balances the send rate with the receive rate. For the synth-100 and synth-1000 applications, the number of messages buffered remains small as long as the cost of the buffered path remains below the send rate.

The send rate we used in this experiment is very high compared to the benchmarks listed in Table 7-1, which communicate only once every 615 cycles (for `barrier`) to 14,200 cycles (for `LU`). If the cost of the buffered path is too large to support the average communication rate, then applications with few synchronization points will tend to buffer a large percentage of their messages. Because the cost of the buffered path as implemented in FUGU and Glaze is only 232 cycles, the system is able to handle very high sustained message rates while buffering only a small fraction of the total messages.

The limit on buffering is set by the *best-case* throughput of the buffering system, *i.e.,* by the minimum, per-message overhead. As noted in Chapter 2, buffering systems work best on batches of messages rather than single messages because of thread scheduling overhead and cache misses.

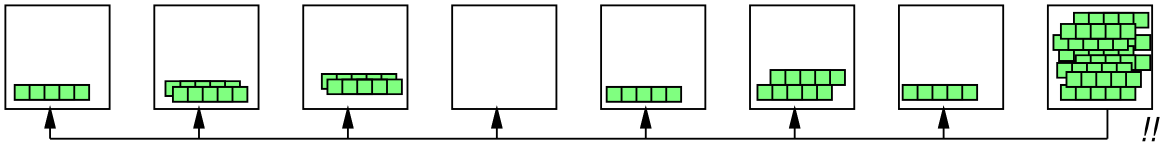


Figure 7-23. Overflow control experiment.

If the buffering system is invoked when the message rate is high, messages quickly become batched which then helps the system escape from buffering.

A buffering system where buffer-insertion is performed in hardware can have a higher throughput in this batch-buffering mode than a system using software buffering because insertion cost is part of the minimum per-message overhead. Compared to a system with hardware buffering, An application on a system with two-case delivery must accept either a lower average message throughput or a rate of synchronization higher than the rate of buffering incidents to avoid runaway buffering. Runaway buffering is handled by overflow control. We examine overflow control next.

7.5 Overflow Control

The previous section shows that demand for buffering in the direct VNI is not an issue for well-behaved applications. There remains the case of ill-behaved applications, including applications under development and applications encountering unexpected conditions at runtime. This problem is unique to systems like the direct VNI that provide virtual buffering and guaranteed delivery. Systems with limited buffering avoid the problem by performing flow control as part of the message protocol. Rather than adding flow control overhead to every message, the direct VNI approach is to provide "overflow control" at a much coarser level via scheduling. This section evaluates an implementation of one overflow control mechanism and policy.

We use a limited version of the overflow control algorithm described in Chapter 5. The paging system in Glaze/PhOS is incomplete; rather than introduce a paging system, we use overflow control with the "low water" mark, N_{low} effectively set to infinitely. The resulting system is subject to deadlock, but is sufficient for our purposes because the test application does not deadlock. To briefly reiterate, the algorithm works as follows:

- The buffer-insert handler of the virtual buffering system for an application compares the number of pages currently in use for buffering, L_{queue}, to a threshold, $L_{threshold}$. If the number of pages in any one process exceeds the the threshold, then the application *globally* switches into an overflow-control mode (Figure 7-23).
- In overflow-control mode, the thread schedulers in each process of the application schedule only threads that tend to consume messages (*i.e.,* the virtual buffering system's cleanup thread).
- The application remains in overflow-control mode until the number of pages in use falls back to zero. At that point, the application globally switches back to its normal mode.

The second network is used to communicate the transition from normal mode to overflow mode since the main network tends to be blocked at the time. The main network is used for all other

Process:	0	1	2	3	4	5	6	7
Threshold				**Pages**				
5	8	7	7	7	7	7	7	10
6	9	8	**23**	9	8	8	8	9
7	10	9	**38**	9	9	9	9	9
8	11	10	10	11	10	10	11	13
9	12	11	**43**	11	12	11	11	11
10	13	12	12	12	17	12	12	12
11	14	13	13	13	13	13	13	13
12	15	14	14	15	14	14	14	14
13	16	15	15	15	15	15	15	15
14	17	16	16	16	**22**	16	16	17
15	18	17	17	17	**30**	17	17	17
16	19	18	18	18	18	18	18	19
17	20	19	19	19	20	19	19	20
18	21	20	20	20	20	20	20	20
19	22	21	21	21	21	21	22	21
...								
∞	495	463	478	469	483	486	483	797

Table 7-5. Overflow control limits the number of pages required to the threshold value with a few isolated exceptions. The table lists the maximum number of pages used over all processors and over the lifetime of the application with overflow control versus the control threshold. A threshold of ∞ corresponds to disabling overflow control.

communication. The implementation is limited in that the mechanism is cooperative in two ways. First, the virtual buffering system and the thread scheduler are implemented in a user-level library that an application can contrive to avoid or change. Second, the mechanism makes assumptions about which threads send messages. In particular, we assume that message handlers do not send (many) messages.

We use a synthetic test application because, as shown in Section 7.4, our real applications do not incur excessive buffering. The test application runs alone, in parallel on all processors. Each process in the application runs a loop that sends messages to a neighboring process. The messages handlers are contrived so that they all induce buffering — they invoke DMA across a page boundary, a case the operating system currently handles by invoking buffering voluntarily, as described in Appendix A.

Overflow control keeps the number of pages required for buffering low. Table 7-5 tabulates the maximum number of pages required per process (L_{queue}) over the lifetime of the application in each experimental run. Each run uses a different threshold, $L_{threshold}$). The features of the table are as follows. First, with overflow control disabled (threshold of ∞), the number of pages required per processor is about 500. This number is a characteristic of the test application. Second, for the bulk of the table, the number of pages required is within about three of the high-water setting, showing pretty good control. The maximum number is higher than the threshold because the overflow control mechanism takes some time to throttle the influx of messages. During that time, more messages can be received. Finally, several data (indicated in boldface) show that occasionally even more pages

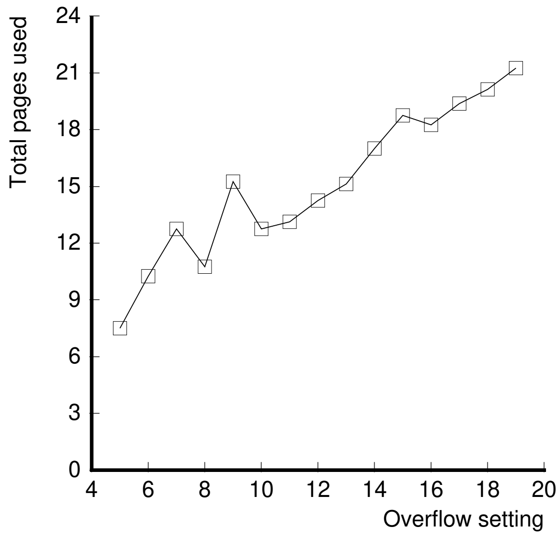


Figure 7-24. A plot of the average of the maximum number pages used over all processors over the lifetime of the application with overflow control versus the control threshold (eight processors).

are required. This effect is possible because the overflow control mechanism itself is implemented at user level, not in the kernel, and thus is subject to the slowdowns from page faults, unfortunate scheduling, etcetera. Any slowdown in the application of the overflow control mechanism leads to more pages in the buffer.

The result is that this overflow control mechanism shows good control over the number of pages required. Figure 7-24 shows how the number of pages required generally follows the threshold. We conclude that overflow control is a promising approach to limiting buffer consumption without introducing protocol overhead into the fast case:

Conclusion 5: Physical buffer consumption can be controlled via nonintrusive overflow control.

Chapter 8

Related Work

The direct VNI builds on work from a number of sources. Likewise, other projects share the goal of combining programmability and protection with performance. This chapter describes the more closely related work in message models, in network interfaces and in techniques that appear in the direct VNI.

8.1 Messaging Models

The UDM model is related to other low-level message models that are intended to serve as building blocks for custom communication at user level within a single protection domain. The Active Messages work [78] gave a name to this style of model which appeared earlier in Mosaic [70], the J-machine [16] and others. UDM is similar to Active Messages and related to Remote Queues (RQ) [8].

UDM shares the Active Message goal of providing a minimal building block and uses the same convention of specifying a handler by a raw procedure address. UDM differs from Active Messages in two important ways. First, UDM codifies explicit control over message delivery (polling or interrupts with user atomicity) as part of the model for full functionality and efficiency. A UDM programmer has the ability to control the communication system with the same flexibility as an in-kernel device driver. Second, while Active Messages defines separate logical request and reply networks and handler types as an anti-deadlock discipline, UDM relies on unlimited buffering to break deadlocks.

The original Active Messages work was on the CM-5 and coexisted with multiprogramming only through strict gang scheduling. Subsequent work on Active Messages broadens its applicability to general multiprogramming [52] by defining indirection tables to safely map handler specifiers to handlers.

UDM is closely related to Remote Queues. The RQ implementation on Alewife used a software version of user-controlled atomicity and the RQ paper outlined a hardware design in progress. Chapter 4 presented the details of that hardware atomicity mechanism here in the context of FUGU. Like RQ, UDM depends on buffering to avoid deadlock rather than on explicit request and reply networks, although some RQ implementations also offer multiple, named queues. Remote queues provide a polling-based view of a network interface with support for system interrupts in critical situations while UDM offers a more general view in which the application freely shifts between polling and user-interrupt modes.

8.2 Network Interfaces

A number of network interface designs have appeared in both research and commercial machines. Some of these NIs have addressed parts of the virtual network interface problem or have attacked the whole problem in ways other that the approach taken by the direct VNI. We'll concentrate on the ones that attack all three parts of the VNI problem. That is to say, direct interfaces with some support for protection, memory-based interfaces that provide performance and hybrids in a similar vein to our direct VNI.

The use of a coprocessor as part of the network interface can be viewed as an issue that is orthogonal to the VNI problem. The role of a coprocessor can be interpreted two ways, either as part of the hardware or as an independent, programmable entity. Simple coprocessors, as in the SP-2 [72] or in *T-Voyager [2], can be viewed as a hardware implementation technique. More elaborate coprocessors, as in the CS-2 [66], Flash [41] or Typhoon [62], that are programmable need a network interface themselves. Either the coprocessor is an opaque hardware component and the VNI problem arises as the processor, or the coprocessor is a user-programmable entity in which case we have to think about the VNI problem at the *coprocessor's* interface to the network.

Protected Direct Interfaces. The direct VNI in FUGU builds on work in direct network interfaces in a number of previous machines [70, 16, 7, 61, 1, 45, 25]. Direct interfaces allow and require the processor to handle messages directly out of the network with minimal buffering. The direct approach maps naturally to a programmable, low-level messaging model and has high performance but has been difficult to protect. A few machines, the CM-5 [45], *T [61] and the M-machine [25] and have provided this kind of interface with protection.

The CM-5 reconciles its direct interface with a restricted form of multiprogramming via partioning, strict gang scheduling and by context-switching the network partitions with the processors. By context-switching the processors and the network together, the CM-5 solves the isolation and undeliverability problems. The CM-5 provides hardware support in the network switches in the form of an "all-fall-down" mode, which allows the kernel to quickly and transparently unload and reload user messages found in the network at the time of the switch. If one application clogs the network with user messages, the network is unloaded before the next application runs (and dutifully reclogged for the original application). The CM-5 has four networks in hardware: request and reply user networks, a "control" network and a debugging network. Strict gang scheduling is a form of multiprogramming but is a severe restriction that hampers the development of, for instance, client-server applications. The CM-5 supports virtual memory with swapping but not with demand paging.

The network interfaces in *T and the M-machine, like the direct VNI hardware, provide protection sufficient for the operating system to demultiplex messages, although operating systems for these machines have not been developed. The direct VNI system could be implemented on these platforms. *T would have included GID checks and a timeout on message handling for protection as in the direct VNI. The M-machine protects a direct interface by restricting message reception to trusted handlers.

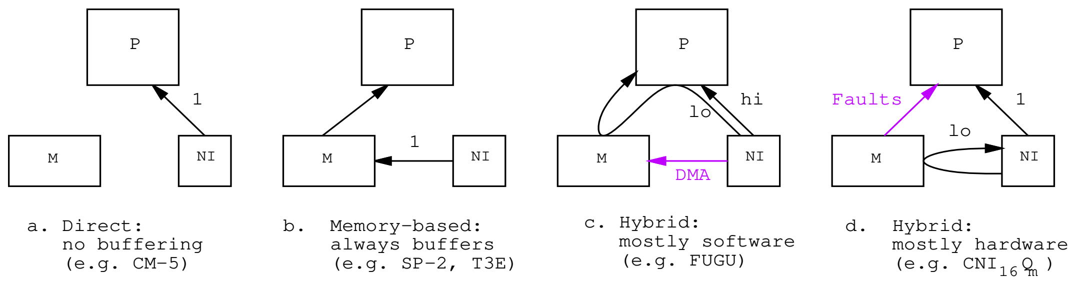


Figure 8-1. Approaches to buffering. The annotations on the arcs represent relative frequencies along each path.

The M-machine's MAP processor is multithreaded, so that the cost of forwarding a message from a trusted handler to a user thread is particularly low. Although the direct VNI approach could be implemented on the M-machine, the M-machine also offers hardware support for a return-to-sender strategy for dealing with undeliverable messages.

Memory-Based Interfaces. Attaching a network interface to the memory system has been an attractive way to provide network access, both in workstations and in many parallel machines, for two reasons. First, attaching the network to the memory system allows standard processors to be used. The memory-based approach will remain attractive until high-performance network interfaces become sufficiently standard to be integrated on-chip in mainstream processors. Second, a bulk transfer mechanism for coarse-grain computing is widely accepted as beneficial while fine-grain mechanisms remain controversial. Since efficient bulk transfer requires DMA, it has been natural to make the network interface design memory-centric with support for small messages a secondary consideration.

Memory-based interfaces in multicomputers [6, 9, 66, 69, 72, 24] and workstations [17, 19, 76, 77] limit the performance of the interface to the speed of memory, but provide easy protection for multiprogramming if the network interface also demultiplexes messages into per-process buffers. Automatic hardware buffering also deals well with sinking bursts of messages and provides the lowest overhead (by avoiding the processors) when buffering is *required* because messages are not handled immediately. Memory-based interfaces could incorporate virtual buffering techniques to expand the receiving buffer automatically. Virtual buffering with a memory-based interface in hardware encounters all of the design challenges described in Chapter 5, albeit with less of a performance requirement since hardware handles the common cases.

Sender-based message systems use network interfaces in memory but allow the sending processor to participate in managing the receiving processor's memory. Remote-memory communication as in SHRIMP [6] and the DEC memory-channel [24] give a sending process the means to directly write memory in another process on another processor. More elaborate sender-based models as in Hamlyn [81, 9] and Hybrid Deposit [59] support more communication options, *e.g.,* the ability to insert into a remote queue, but retain the idea that the sender manages the memory. The sender-based approach is interesting because it exposes the fundamental costs of memory-based communication in its programming model. Like our low-level UDM model, the programmer is given the means to mitigate the costs of communication. The sender-based approaches must contend with memory management, but mechanisms are exposed for providing custom memory management tailored to the application.

Hybrid Interfaces. The direct VNI employs a hybrid approach to network interface design that uses both a direct interface for speed and provides buffering for convenience. Other interfaces take a similar approach, notably Wisconsin's CNI [56, 57] and the descendants of *T, *T-NG [11] and *T-Voyager [2]. Figure 8-1 (identical to Figure 2-3) gives a schematic view of different approaches to message delivery. Parts (a) and (b) show direct and memory-based delivery, respectively. Part (c) represents the direct VNI approach in which the hardware is direct but the message system supports two paths. Part (d) represents an alternate approach, taken in the $\mathrm{CNI}_{16}\mathrm{Q}_m$, in which the network interface provides both a fast path and a (potentially virtual) buffered path by using the network interface hardware to manage messages. The processor accesses the network through the CNI directly, but the CNI hardware also has the ability to transparently spill its internal buffer to memory as necessary.

The CNI approach is potentially hardware-intensive, for instance requiring a duplicate translation cache in the network interface. The direct VNI uses operating system software to initiate buffering and uses a DMA engine shared with its bulk transfer mechanism to move the message data. The hardware requirements are kept minimal for on-chip implementation and amount to a small, single message queue and a simple DMA engine. A range of implementation options exist between the CNI and the direct VNI, however. For instance, the translation cache could be minimal, storing only the physical addresses of a few pages actively in use for buffering.

A subtle difference between the $\mathrm{CNI}_{16}\mathrm{Q}_m$ and the direct VNI as each is currently implemented is in when they choose to initiate buffering. The $\mathrm{CNI}_{16}\mathrm{Q}_m$ apparently switches to buffering in memory whenever the small buffer in the NI is full. [58] In contrast, the direct VNI requires some other event (like a timeout) to enable a switch to software buffering. The hardware support for buffering in the $\mathrm{CNI}_{16}\mathrm{Q}_m$ makes it relatively lower cost to switch whereas software buffering in the direct VNI is relatively more expensive. Again, a range of implementation options exist and the best decision about when to switch is likely to be in between these two extremes.

A different hybrid approach is taken in the *T-NG [11] system and the subsequent *T-Voyager system [2]. In *T-NG, the network interface hardware demultiplexes incoming messages into one of several moderate-sized hardware queues implemented as dual-ported RAM on the L2 cache bus. The multiple queues allow a limited number of applications to be active simultaneously. Additional applications can be multiplexed onto these queues by negotiating for which queues should be active or "cached". *T-Voyager implements the same idea of cached queues using a shared buffer on the system bus of an SMP and a coprocessor to manage the queue insertions. Like the direct VNI, *T-Voyager overflows its queues to memory if necessary. Unlike the direct VNI, *T-Voyager can use its coprocessor to perform this work.

8.3 Miscellaneous

Techniques used in our virtual buffering system are related to several other systems. The Active Message implementation in SUNMOS [63] on the Intel Paragon uses kernel code to unload the message interface and to queue messages to be handled by a user thread. The SUNMOS approach corresponds to using the software-buffered path in UDM continuously.

Fbufs [18] are an operating-system construct used to efficiently feed streams of data across protection domains. The UDM virtual buffering system employs similar techniques in a specialized implementation to manage its buffer memory.

Network overflow in Alewife [40] is a form of two-case delivery used for a restricted purpose. Alewife supports distributed shared memory in hardware using a single network. Network overflow uses software buffering to simulate infinite buffering in the network for the purpose of breaking deadlocks in the shared-memory protocol.

The Polling Watchdog [53] integrates polling and interrupts for performance improvement. The resulting programming model is interrupt-based in that application code may receive an interrupt at any point; the application cannot rely on atomicity implicit in a polling model. A polling watchdog uses a timeout timer on message handling to accelerate message handling if polling proves sluggish. The direct VNI hardware includes an identical timer but uses it only to let the operating system clear the network. A polling watchdog mode could be implemented in the direct VNI system.

Chapter 9

Conclusion

Scalable workstations are a reasonable vision of near-future servers. Scalable workstations combine the scalable performance of MPP communication mechanisms and hardware with protection for mixed, multiprogrammed workloads. The challenge in scalable workstations is to integrate the communication mechanisms with standard workstation protection features. This integration, captured in the virtual network interface problem, is difficult because efficiency comes from tight coupling of network hardware with applications but multiprogramming tends to interfere with that tight coupling. A good virtual network interface should solve three problems:

1. Programmability: the programmer's model must be efficient.
2. Protection: the implementation must permit multiprogramming and virtual memory.
3. Performance: the communication implementation must be fast.

Conventional approaches sacrifice either the flexibility of the communication model, the flexibility of multiprogramming, or the performance of the interface. The direct virtual network interface approach presented in this thesis preserves all three goals.

The direct virtual network interface solution starts with an efficient, low-level model that gives the user-level programmer kernel-like control over messages. The implementation meets the other two goals using the novel techniques of two-case delivery and virtual buffering. Two-case delivery provides a fast case implemented in hardware and a robust case using buffering in software. The combination provides the speed of hardware with the flexibility of buffering. Virtual buffering allows the fast case to be faster by guaranteeing delivery, eases programming by removing buffer space limitations from the programmers model and and additionally allows the system to manage buffering resources automatically.

The evaluation presented in this thesis makes two major points. First, experiments show that the fast case is close in speed to unprotected hardware and that it remains the common case under almost every set of operating conditions. Second, we show that the problem of potentially unbounded buffer consumption due to the combination of guaranteed delivery with unacknowledged messages is not a problem for ordinary applications and can be controlled when it occurs in unusual applications or situations.

In terms of the three issues articulated in the problem statement:

1. The UDM programming model is an efficient target for the application programmer, a library developer or the compiler. User messages correspond one-to-one with hardware messages and the application has the opportunity to customize its protocols.

2. The direct VNI system is compatible with multiprogramming and demand-paged virtual memory. The two-case delivery and virtual buffering techniques make multiprogramming slightly less flexible than a system with hardware buffering because the scheduler may need to coschedule more often. However, as noted in Section 7.3, part of the increased demand for coscheduling comes from lower latency.

3. Latency and bandwidth in the direct VNI are good – close to those in an unprotected, kernel-level interface – provided the interface is kept in the fast case. Because of software buffering, the direct VNI is slightly less tolerant of borderline programs that send streams of unacknowledged messages compared to a system with hardware buffering. While all such programs need to add flow control as message rates increase, the software-buffered approach requires flow control at a slightly lower maximum rate.

We conclude that the architectural techniques of two-case delivery and virtual buffering in a direct virtual network interface make a tightly-coupled network interface viable in a multiuser multiprocessor. As shown by the plot in Figure 7-11, the direct VNI allows FUGU to achieve good parallel performance because only $14 - 33\%$ of messages are buffered in most of our applications while 10% of CPU time is devoted to interactive tasks. As parallel processing becomes more mainstream and as increased system integration moves more components on-chip, the advantages of the direct virtual network interface approach only become more important.

Appendix A

Bulk Transfer

This appendix provides details about the bulk transfer mechanism in FUGU. The focus of the direct virtual network interface is on support for small messages. There is, however, a related bulk transfer mechanism using that uses DMA for high bandwidth. The DMA implementation follows the ideas in [50], but is incomplete.

Model. Bulk transfer is integrated with the UDM model as an option on the `inject` and `extract` operations defined in Section 3.1. At the send side, `inject` with DMA looks like:

`injectdma` (*header, handler, word0, word1, . . . , srcaddr, len*)

where *srcaddr* and *len* are the address and length of a block to be appended to the message. The block is simply appended: the receiver cannot tell how the message was constructed. At the receive side, `extract` with DMA looks like:

`extractdma` (*nwords, dstaddr, len*) $\Rightarrow$ *(header, handler, word0, word1, . . .*)

where *nwords* is the number of words at the beginning of the message to be extracted in the usual way and *dstaddr* and *len* describe a block in memory to receive the data from the remainder of the message.

DMA proceeds asynchronously after the `injectdma` or `extractdma` operation is started. Completion notification is restricted to polling on *injectdma-done* and *extractdma-done* flags. These flags return true when the last `injectdma` or `extractdma`, respectively, has completed.

Library Implementation. The bulk transfer primitives are accessed by the applications in Chapter 7 using a variation on the `do_on` statement and handler declarations introduced in Section 6.3. The DMA version of `do_on` looks like:

`do_on_dma` (*dst, handlername [, arg0 [, . . . [, arg3]]], addr, ndwords*)

The DMA block address, *addr*, must be doubleword aligned. The *ndwords* argument must be a compile-time constant. The DMA block must not cross a page boundary. The total size of the message must be less than a page (4096 bytes). The `do_on_dma` statement translates into a table lookup for the *dst* node number, as in `do_on`, a TLB probe operation to translate *addr* followed by store of the resulting physical address to the CMMU, and then an `injectdma` operation. The handling of the physical address from the TLB probe operation is a security hole that we accept in the prototype. A real implementation would pass the physical address securely from the TLB to the DMA engine, for instance using the technique described by Heinlein in [27].

```
void dma_handler(handlername [, arg0 [, ... [, arg3]]], addrexp, lenexp)
{
   ⟨atomic handler code⟩
   user_active_global();
   ⟨thread code⟩
}
```

The DMA block address in the handler, *addrexp*, and the block length, *lenexp* are expressions in terms of variables available at the receiver, including *arg0* through *arg3* if given. The address must be doubleword aligned. The length is in bytes and must be a multiple of doublewords. The DMA block is logically permitted to cross page boundaries, but the implementation switches voluntarily to buffering mode in this case so it isn't very efficient. The total size of the message must be less than a page.

Two-Case Delivery and Virtual Buffering. Bulk transfer messages are treated the same way as small messages: the entire message is copied into the software buffer in buffering mode. Copying large blocks is a poor idea that is worth some extra effort to avoid. There are a couple of possible options that fall between fast mode and the slow mode, much like cross-domain upcalls in Figure 3-5.

First, as suggested in [50], `extractdma` operations that specify a destination block in an inaccessible page could write to a freshly allocated page and "patch up" the differences afterward. A page can be inaccessible because it is a virtual memory page that is swapped out or because is it a shared memory page and portions of it are cached remotely in a write-only state. This two-phase write-and-clean-up action is possible because the coherence semantics of DMA are explicit: the DMA'd data is not considered readable until the `extractdma-done` flag is true.

A second option is more intrusive. Currently all messages handlers are executed in order whether they are handled via the fast path or slow path. An exception could be made for some or all DMA blocks, however. For such a block received in buffer mode, the buffer insertion code could attempt to interpret enough of the handler to determine where the DMA block is intended to be stored.

Efficient bulk transfer in a system like FUGU is one interesting direction for future work.

References

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

[2] Boon S. Ang, Derek Chiou, Larry Rudolph, and Arvind. Message Passing Support on StartT-Voyager. CSG Memo 387, Computation Structures Group, MIT Laboratory for Computer Science, July 1996.

[3] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

[4] Robert C. Bedichek. Talisman: Fast and Accurate Multicomputer Simulation. In *Proceedings of the 1995 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

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

[7] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 70–81, June 1990.

[8] Eric Brewer, Fred Chong, Lok Liu, Shamik Sharma, and John Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 1995.

[9] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 245–259, 1996.

[10] John B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency*. PhD thesis, Rice University, August 1993.

[11] Derek Chiou, Boon S. Ang, Arvind, Michael J. Beckerle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. START-NG: Delivering Seamless Parallel Computing. In *Proceedings of the 1st International Conference on Parallel Proceessing (Euro-Par '95)*, August 1995. Also available as CSG Memo 371, Computation Structures Group, MIT Laboratory for Computer Science.

[12] Frederic T. Chong, Rajeev Barua, Fredrik Dahlgren, John D. Kubiatowicz, and Anant Agarwal. The Sensitivity of Communication Mechanisms to Bandwidth and Latency. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[13] Compaq Computer Corporation. Virtual Interface Architecture for SANs. Technology Brief, May 1997. Document Number 508A/0597.

[14] Alan L. Cox, Sandhya Dwarkadas, Honghui Lu, and Willy Zwaenepoel. Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers. In *Proceedings of the 11th International Symposium on Paralel Processing*, April 1997.

[15] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, 1991.

[16] William J. Dally et al. The J-Machine: A Fine-Grain Concurrent Computer. In *Proceedings of the IFIP (International Federation for Information Processing), 11th World Congress*, pages 1147–1153, New York, 1989. Elsevier Science Publishing.

[17] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, pages 36–43, July 1993.

[18] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, December 1993.

[19] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of the Conference on Communication Architectures, Protocols and Applications*, pages 2–13, 1994.

[20] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[21] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, October 1–5, 1996.

[22] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. In *Journal of Parallel and Distributed Computing*, pages 306–318, December 1992.

[23] Edward W. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1993.

[24] Marco Fillo and Richard B. Gillett. Architecture and Implementation of Memory Channel 2. *Digital Technical Journal*, 9(1):27–41, 1997.

[25] Marco Fillo, Stephen W. Keckler, W.J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156. IEEE Computer Society, November 1995.

[26] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1994.

[27] John Heinlein, Kourosh Gharachorloo, and Anoop Gupta. Integrating Multiple Communication Paradigms in High Performance Multiprocessors. Technical Report CSL-TR-94-604, Stanford, February 1994.

[28] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[29] Robert W. Horst. TNET: A Reliable System Area Network. *IEEE Micro*, pages 37–45, February 1995.

[30] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[31] Kirk Lauritz Johnson. *High-Performance All-Software Distributed Shared Memory*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, December 1995. Also available as Technical Report LCS-TR-674, MIT Laboratory for Computer Science.

[32] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, December 1997.

[33] Vijay Karamcheti and Andrew Chien. Do Faster Routers Imply Faster Communication? In *Proceedings of Parallel Computer Routing and Communications Workshop*, May 1994.

[34] Vijay Karamcheti and Andrew A. Chien. View Caching: Efficient Software Shared Memory for Dynamic Computations. In *Proceedings of the 11th International Symposium on Parallel Processing*, April 1997.

[35] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[36] Kendall Square Research. KSR-1 Technical Summary, 1992.

[37] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michał Cierniak, Srinivasan Parthasarathy, Wagner Meira, Jr., 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*, pages 157–169, June 1997.

[38] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory; Early Experience. In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[39] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Conference on Supercomputing*, pages 195–206, July 1993.

[40] John D. Kubiatowicz. *Integrated Message-Passing and Shared-Memory Communication in the Alewife Multiprocessor*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1998.

[41] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

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

[43] Victor Lee. An Evaluation of Fugu's Network Deadlock Avoidance Strategy. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1996.

[44] Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph. Implications of I/O for Gang Scheduled Workloads. In *Workshop on Parallel Job Scheduling, IPPS '97*. Springer Verlag, 1997.

[45] Charles E. Leiserson, Aahil S. Abuhamdeh, and David C. Douglas et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[46] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[47] Kevin Lew, Kirk Johnson, and Frans Kaashoek. A Case Study of Shared-Memory and Message-Passing Implementations of Parallel Breadth-First Search: The Triangle Puzzle. In *Third DIMACS International Algorithm Implementation Challenge Workshop*, October 1994.

[48] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Computing*, pages 94–101, 1988.

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

[50] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and M. Frans Kaashoek. FUGU: Implementing Protection and Virtual Memory in a Multiuser, Multimodel Multiprocessor. Technical Memo MIT/LCS/TM-503, October 1994.

[51] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Walter Lee, Victor Lee, Anant Agarwal, and M. Frans Kaashoek. Exploiting Two-Case Delivery for Fast Protected Messaging. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[52] Alan Mainwaring. Active Message Application Programming Interface and Communication Subsystem Organization. UC Berkeley, Computer Science Department, December 1995.

[53] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin Theobald, and Xin-Min Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 179–188, May 1996.

[54] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995.

[55] Jonathan E. Michelson. Design and Optimization of Fugu's User Communication Unit. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1996.

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

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

[58] Subhendu S. Mukherjee and Mark D. Hill. The Impact of Data Transfer and Buffering Alternatives on Network Interface Design. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[59] Randy Osborne. A Hybrid Deposit Model for Low Overhead Communication in High Speed LANs. Technical Report 94-02v3, MERL, 201 Broadway, Cambridge, MA 02139, June 1994.

[60] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[61] Gregory M. Papadopoulos, G. Andy Boughton, Robert Greiner, and Michael J. Beckerle. *T: Integrated Building Blocks for Parallel Computing. In *Supercomputing '93*, pages 624–635, November 1993.

[62] Steve 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 325–336, April 1994.

[63] Rolf Riesen, Arthur B. Maccabe, and Stephen R. Wheat. Split-C and Active Messages under SUNMOS on the Intel Paragon. Unpublished, April 1994.

[64] Marcel-Cătălin Roşu, Karsten Schwan, and Richard Fujimoto. Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach. In *Proceedings of the 6th IEEE International Symposium on High-Performance Distributed Computing*, 1997.

[65] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1–5, 1996.

[66] Klaus E. Schauser and Chris J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Symposium on Parallel Processing*, 1995.

[67] 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 *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[68] Ioannis Schoinas and Mark D. Hill. Address Translation Mechanisms in Network Interfaces. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[69] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.

[70] C.L. Seitz, N.J. Boden, J. Seizovic, and W.K. Su. The Design of the Caltech Mosaic C Multicomputer. In *Research on Integrated Systems Symposium Proceedings*, pages 1–22, Cambridge, MA, 1993. MIT Press.

[71] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, pages 5–44, March 1992.

[72] Marc Snir and Peter Hochschild. The Communication Software and Parallel Environment of the IBM SP-2. Technical Report IBM-RC-19812, IBM, IBM Research Center, Yorktown Heights, NY, January 1995.

[73] P. G. Sobalvarro and W. E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Lecture Notes in Computer Science, 949*, pages 106–126. Springer Verlag, 1995. Workshop on Parallel Job Scheduling, IPPS '95.

[74] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[75] Patricia Jane Teller. Translation-Lookaside Buffer Consistency in Highly-Parallel Shard-Memory Multiprocessors. Technical Report RC 16858, IBM, Yorktown Heights, NY, May 1991.

[76] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Efficient Support for Multicomputing on ATM Networks. Technical Report UW-CSE-93-04-03, University of Washington, Seattle, WA, April 1993.

[77] 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 Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

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

[79] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu, Thomas Wicki, and Winfried Wilcke. The Mercury Intercoonect Architecture: A Cost-Effective Infrastructure for High-Performance Servers. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 98–107, June 1997.

[80] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating memory management into user-level network interfaces. In *Proceedings of Hot Interconnects IV*, 1997.

[81] John Wilkes. Hamlyn — An Interface for Sender-Based Communications. Department Technical Report HPL-OSR-92-13, HP Labs OS Research, November 1992.