To appear in the Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, February, 1998

Exploiting Two-Case Delivery for Fast Protected Messaging

Kenneth Mackenzie,[*] John Kubiatowicz,[†] Matthew Frank, Walter Lee,
Victor Lee,[‡] Anant Agarwal and M. Frans Kaashoek
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{kenmac,kubitron,mfrank,walt,wklee,agarwal,kaashoek}@cag.lcs.mit.edu

Abstract

We propose and evaluate two complementary techniques to protect and virtualize a tightly-coupled network interface in a multicomputer. The techniques allow efficient, direct application access to network hardware in a multiprogrammed environment while gaining most of the benefits of a memory-based network interface. First, **two-case delivery** *allows an application to receive a message directly from the network hardware in ordinary circumstances, but provides buffering transparently when required for protection. Second,* **virtual buffering** *stores messages in virtual memory on demand, providing the convenience of effectively unlimited buffer capacity while keeping actual physical memory consumption low. The evaluation is based on workloads of real and synthetic applications running on a simulator and partly on emulated hardware. The results show that the direct path is also the common path, justifying the use of software buffering. Further results show that physical buffering requirements remain low in our applications despite the use of unacknowledged messages and despite adverse scheduling conditions.*

1 Introduction

Research in the past decade has seen the speed of communication mechanisms such as coherent shared memory and message passing increase to the point that each can be engineered to within a small factor of the speed of a local memory miss. Such fine-grain communication mechanisms have in turn enabled a much larger class of applications based on either shared memory or message passing programming styles to benefit from parallel processing. Shared-memory communication mechanisms extend naturally to multiprogrammed environments, with virtual memory based isolation of processes as the basis for protection. Protecting fast message interfaces without sacrificing performance, however, is more challenging.

Message passing network interfaces (NIs) 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 [14]. For example, the CNI paper [24] showed that a direct, cache-level interface exhibited 50% higher bandwidth than their best interface placed on the memory bus. As discussed further in Section 2, direct interfaces are challenging to protect without sacrificing efficiency or seriously impairing the multiprogramming model. One appeal of memory-based interfaces is that they may be protected through standard memory mapping mechanisms.

In this paper, we investigate how to provide protected message passing with a low-overhead direct network interface. The key challenge is to efficiently virtualize and protect both access to network interface data and control over delivery of network events. Specifically, for performance, an application receives a message via polling or user interrupt and pulls the message directly from the network fabric at cache speeds without traversing main memory. In a multiprogrammed environment, the challenge is complicated by cases where a message arrives when the target application is not scheduled, where a message handler runs but takes a page fault and where a message handler blocks the network for an extended period. Our approach is to handle the common, direct-delivery case with the efficiency of unprotected, kernel-level access while handling the exceptional cases in a manner that at worst leads to graceful degradation in performance. Our protected message reception path allows reception of a null message at user level via interrupt in 87 cycles compared to 54 cycles at kernel level without protection. Similarly, we receive a null message via polling in 9 cycles at both user or kernel levels. These times are of the same order of magnitude as a cache refill time from DRAM in a modern processor.

We propose two techniques to address the problem of protecting a tightly coupled interface. The techniques employ a combination of software with a small amount of hardware:

- First, *two-case delivery* provides a uniform mechanism for dealing with an application that cannot immediately receive a message. Two-case delivery switches transparently between a direct mode and a software-buffered mode of message delivery. The interface resorts to the buffered mode when a message cannot be delivered safely or in a timely manner to the application. Thus, messages need not be dropped merely for protection reasons.
- Second, *virtual buffering* maintains the software buffer in virtual memory (of the communicating application) allocated on demand. Virtual buffering, like virtual memory, allows the operating system to manage and minimize physical resource consumption while giving the application the illusion of a very large resource. Virtual buffering is applicable to any message system that employs buffering.

The context in which we evaluate our techniques is that of an interface that supports a *User Direct Messaging* (UDM) model. Simi-

[*] Current affiliation: Georgia Institute of Technology, Atlanta, GA 30332
[†] Current affiliation: University of California at Berkeley, Berkeley CA 94720
[‡] Current affiliation: Intel Corporation, Santa Clara, CA 95052

lar to Active Messages [35], UDM defines a lightweight messaging discipline in which every incoming message invokes a user handler. The UDM discipline differs from others by *codifying* explicit control over handler atomicity as part of the user model and thus provides for a smooth integration of interrupt- and polling-driven operation. User-level code thus enjoys the same level of control over the interface as an in-kernel device driver.

We have implemented our techniques in the FUGU multiprocessor system [21, 22]. The FUGU hardware is based on extensions to the (single-user) Alewife machine [1]. We have constructed both a fast simulator and emulated hardware platforms. The emulator extends the Alewife chipset with a modified Cache and Memory Management Controller (CMMU) gate array and an auxiliary FPGA that implement protection and virtualization features for the direct delivery case. The FUGU operating system, called Glaze, is a custom, multiuser operating system based on the Exokernel [16] and supports virtual buffering. The new hardware, with the modified CMMU and the FPGA, has been implemented and currently runs a subset of our applications on two nodes. Most of our results are from the simulator.

We performed experiments using a multiprogrammed workload of real and synthetic benchmarks. Our experiments show that the fast case is indeed the common case and that application characteristics tend to keep the real memory required for buffering low even under adverse scheduling conditions.

The main contributions of this paper are: (a) the notion of two-case delivery, (b) the notion of virtual buffering, (c) our specific implementation of a virtualized interrupt disable mechanism and (d) the UDM model itself. The paper also reports on an implementation of an operating system supporting protected direct messages, and provides early evidence using multiprogrammed workloads that (1) the fast case of direct delivery can indeed be the common case, (2) although virtual buffering provides the guarantee of unlimited buffering, the physical memory requirements are commonly small.

The remainder of the paper is organized as follows. Section 2 puts our work in perspective by describing related work. Section 3 presents the UDM model which forms the framework in which we evaluate our ideas. Section 4 lays out the architecture for our protected implementation of the model using two-case delivery with virtual buffering and provides raw performance data for the two delivery cases. Section 5 describes the results of experiments with applications and our operating system running on a simulator that give the performance of virtual buffering. Section 6 concludes.

2 Related Work

Recent architectures demonstrate emerging agreement that it is important to provide support for efficient, fine-grain message passing, even in conjunction with hardware support for shared memory [1, 2, 13, 18, 27, 30]. The trend in message interfaces has been to reduce end-to-end overhead by providing user access to the interface hardware. We build on previous work in messaging models and mechanisms.

Model. The UDM model is similar to Active Messages [35] and related to Remote Queues (RQ) [6] as an efficient building-block for messaging within a protection domain. UDM differs from Active Messages in that it includes explicit control over message delivery for efficiency. RQ provides 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. The RQ implementation on Alewife used a software version of user-controlled atomicity and the RQ paper outlined a hardware design in progress. We present 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.

The Polling Watchdog [23] 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 FUGU 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 FUGU system.

Direct Network Interfaces. Several machines have provided direct network interfaces. These include the CM-5, the J-machine, iWarp, the *T interface, Alewife, and Wisconsin's CNI [20, 8, 5, 26, 1, 24]. 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 work shows how to partly compensate for a more distant NI by exploiting standard cache-coherence schemes.

Direct interface designs have mostly ignored issues of multiprogramming and demand paging. The CM-5 provides restricted multiprogramming by strict gang scheduling and by context-switching the network with the processors. The *T NI [26] would have included GID checks and a timeout on message handling for protection as in FUGU. The M-machine [12] receives messages with a trusted handler that has the ability to quickly forward the message body to a user thread.

Memory-Based Interfaces. Memory-based interfaces in multicomputers [4, 7, 29, 30, 32] and workstations [9, 11, 33, 34] 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) and when latency is not an issue. Increased integration of computer systems and the mainstreaming of parallel processing challenges both of these assumptions: on-chip network interfaces can have low overhead and parallel programs frequently require coordinated scheduling for predictable, low latencies [3]. FUGU provides low latency for applications where latency matters while including low-cost and reasonably efficient buffering as a fallback mode. Bulk data transfers are handled by a separate direct memory access (DMA) mechanism in FUGU [21].

Hybrids. Our UDM implementation employs both a direct interface for speed and provides buffering for convenience. Other projects share the same goals. Figure 1 gives a schematic view of the different approaches to message delivery. Parts (a) and (b) show direct and memory-based delivery, respectively. Parts (c) and (d) depict hybrid schemes with different approaches. The memory based CNI interface ($\mathrm{CNI}_{16}\mathrm{Q}_m$) [24, 25] provides both a fast path and a (potentially virtual) buffered path by using the network interface to buffer messages. This approach is hardware-intensive, for instance requiring a duplicate translation cache in the network interface. The UDM implementation 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.

The *T-Voyager system [2] represents an intermediate hybrid. In *T-Voyager, the network interface demultiplexes incoming messages into several moderate-sized hardware queues. The multi-

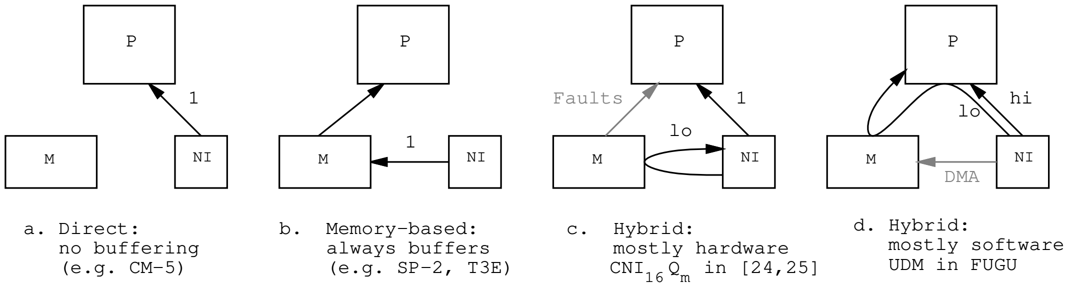

Figure 1: Approaches to buffering. The annotations on the arcs represent relative frequencies along each path

ple queues allow multiple applications to be active simultaneously. Like UDM, *T-Voyager overflows its queues to memory if necessary.

Techniques used in our virtual buffering system are related to several other systems. The Active Message implementation in SUNMOS [28] 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 [10] 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.

3 UDM Model

This section describes the UDM model in abstract terms. The model has two goals. First, it is an efficient 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 on other primitives, UDM is sufficiently low-level to be implemented efficiently in hardware.

UDM provides all of the facilities commonly desired from fast message interfaces within a multiprocessor: low-overhead message construction and launch, as well as reception via interrupts or polling. Part of its simplicity with respect to multiuser environments is that UDM allows the programmer to view the network hardware as a dedicated, user-level resource with infinite buffering. It is up to the hardware and runtime system to maintain this illusion.

UDM has two major components: 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. This 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.

Messaging Model. 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 is an optional handler address, as used in Active Messages. Remaining words represent the data payload and are unconstrained.

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 [17]. Message injection can thus be viewed in the following fashion:

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

If resource contention prevents the network from accepting a given message, the corresponding `inject` operation blocks until successful. Alternatively, 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. 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()` ⇒ *(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 buffering in later sections. In this sense, UDM is a generalization of Remote Queues.

Atomicity Model. UDM assumes an execution model in which one or more *threads* run on each processor. Further, UDM defines a message notification interface consisting of a message available flag, readable by the application, a message available interrupt, described below, and an interrupt disable flag, writable by the application. The interrupt disable flag permits an application to request *atomicity* with respect to message available interrupts. Periods of execution in which interrupts are disabled are called *atomic sections*. When interrupts are disabled, notification is entirely through the message available flag. In this mode, the currently running thread must poll the message available flag and extract messages with `extract` as they arrive.

In contrast, when interrupts are enabled, the existence of an input message causes the current thread to be suspended and an independent *handler*, to be initiated. The sequence of events is illustrated in Figure 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. 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. In

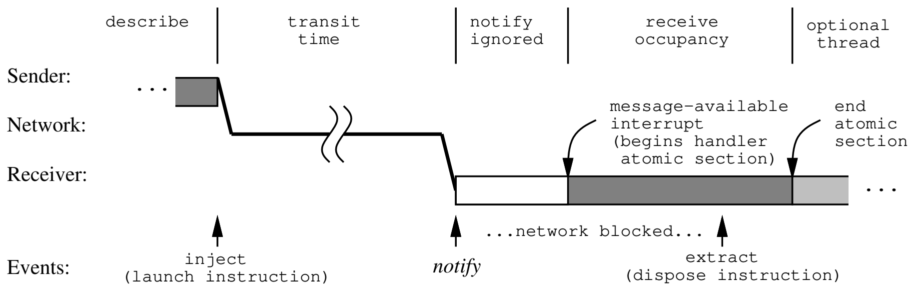

Figure 2: Message time line for interrupt delivery on the fast path.

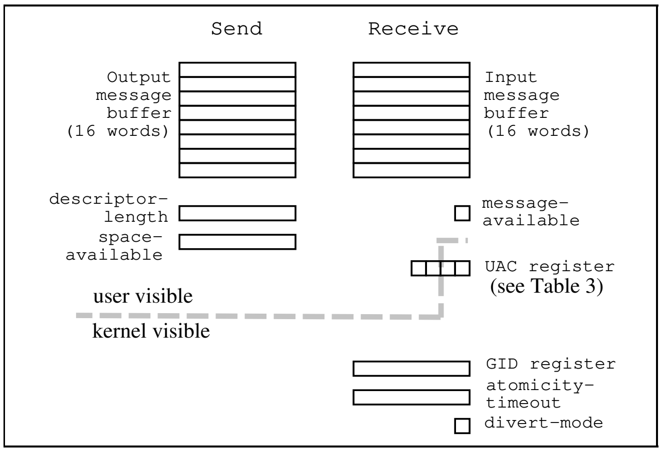

Figure 3: FUGU network interface registers.

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 and 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)`	*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** *UAC* **:=** (*UAC* $\wedge$ ($\sim$`MASK`))

Table 1: FUGU operations

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.

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 mechanism is an *abstraction* for interrupt disable: in the common case, the user's requests for atomicity interact directly with the network interface to defer interrupts. During page faults or other unexpected events, however, these requests are virtualized to free up the physical interface while maintaining the illusion of atomicity to the user. The next section describes how this illusion is implemented.

4 Implementing Two-Case Delivery

This section details the design of the FUGU implementation of the UDM model, showing how it uses two-case delivery to provide user-level access to data and user-level atomicity control as well as support for virtual buffering. The UDM model provides an abstraction of communication. The send-side of the UDM abstraction is implemented in hardware in FUGU. 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.

This section is organized around the parts of two-case delivery. First, we describe 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 control over atomicity for user interrupts and direct polling. Second, we describe the virtual buffering delivery case, we show how it provides semantics identical to the fast case and we introduce an overflow control mechanism used to limit buffer usage of unruly programs. Third, we put the two cases together by describing how the interface to the two modes is kept transparent and how transition is invoked.

4.1 Direct Access Path

The FUGU network interface consists of a set of memory mapped registers shown in Figure 3, a set of atomic operations listed in Table 1 and a set of interrupts and traps listed in Table 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 [17]. To send a message, an application first writes all of the

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 2: FUGU Interrupts and traps

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 1.
atomicity-extend	Requests an *atomicity-extend* trap. See `endatom` in Table 1.

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

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 [21] which is beyond the scope of this paper.

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` 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.

FUGU 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, 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*.

- 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, we would like the user to enjoy similar efficiency in the common case to the operating system, extracting messages directly from the network interface.

We solve these problems by implementing atomicity through a *revocable interrupt disable* mechanism. The main idea behind this mechanism is that we allow the user to *temporarily* disable hardware message interrupts. As long as the network continues to make forward progress, we allow the user to continue disabling interrupts. Should a message stay blocked at the input queue for too long, we *revoke* 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

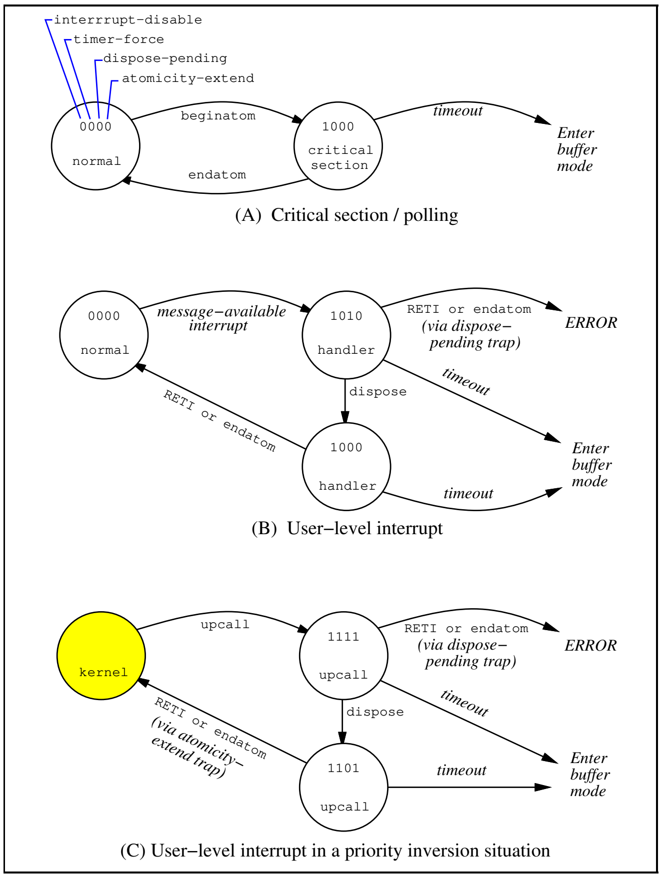

Figure 4: Three revocable interrupt disable examples. User-level nodes are labeled with the *UAC* state at the top. Kernel-level nodes are shaded.

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 we 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 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 1 (but generally whenever either of the kernel bits is set), `endatom` executed in user mode will trap to return control to the operating 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 3.

The use of the revocable interrupt disable mechanism is best illustrated by example. Figure 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).

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: 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.

Fast Path Performance. Table 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 cost is for a blocking `inject` operation. The interrupt-based receive cost represents the basic fast path cost. 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), but we also predict the performance we expect 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.

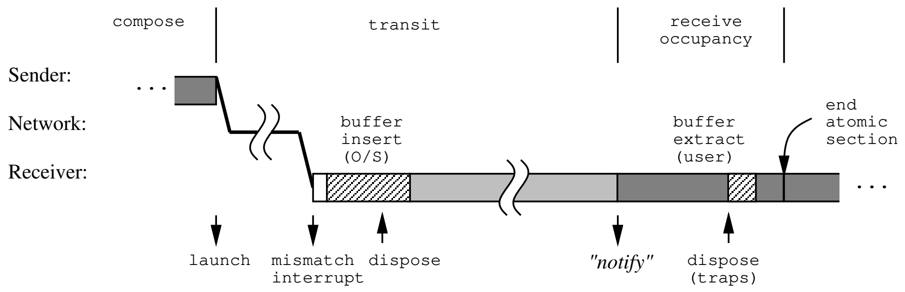

Figure 5: Message timeline for the buffered path.

4.2 Virtual Buffering Path

Virtual buffering allows messages to be buffered in order to preserve the semantics of the UDM model in the face of uncommon but unpredictable cases. The objectives of virtual buffering are to provide:

1. Identical semantics to the fast case using memory for data access and user thread manipulations for atomicity control.
2. Automatic management of physical memory resources.
3. Guaranteed delivery, which helps to avoid application deadlock and allows the fast case to avoid all buffer management overhead.

We address the objectives as follows. First, hardware and software mechanisms and software conventions provide UDM semantics. Second, the software buffer itself is virtualized, allowing essentially unlimited buffering while avoiding dedicating physical memory to what is in practice an infrequent case. Finally, delivery is guaranteed within the limits of swap space by providing a deadlock-free path to backing store. An overflow control mechanism makes unlimited virtual buffering practical by providing feedback from the buffering system to the system scheduler. The design details are discussed below along with low-level performance. The interaction of buffering with application characteristics is discussed in Section 5.

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.

In the steady state, illustrated as a timeline in Figure 5, buffered mode is supported by the *divert-mode* bit in the network interface (Figure 3). When *divert-mode* is set, all incoming messages cause kernel *mismatch-available* interrupts. The *mismatch-available* interrupt handler in the operating system demultiplexes incoming messages into the software buffer of the application indicated by the GID in the message header.[1] In addition, *divert-mode* set causes the user-mode `dispose` instruction 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.

Guaranteed Delivery. The buffering system needs to be able to provide buffer space to absorb incoming messages rapidly. However, it is inconvenient to pin down physical pages in order to serve what we expect to be an infrequent case. Virtual buffering stores messages in virtual memory and allocates physical page frames to back that virtual memory only on demand.

The interrupt/trap handler 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 buffer insertion code only stalls when there are absolutely no page frames available on the node. Without additional mechanism, this stall situation would lead to deadlock. We propose to rely on an extra logical network to avoid deadlock in the worst cases and to use scheduling techniques to preserve efficiency in most cases. We have only partially implemented the deadlock avoidance and scheduler feedback techniques at this time, but foresee no fundamental problems with the following approach:

- We avoid deadlock at the lowest level by arranging for key services to rely on a second logical network reserved to the operating system as a 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 [8] in the main network, a LAN or a service network would serve the purpose. Our emulator hardware provides a very simple, bit-serial network.
- 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 of virtual buffering space corresponds to

[1] We don't actually use the processor to copy the message into memory; there is a DMA mechanism that can be optionally invoked as part of the `dispose` operation.

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 5.2), so the buffering system advises the scheduler to gang schedule the application.

Buffering Performance. The existence of the buffered path with guaranteed delivery improves performance by enabling the fast path to avoid all buffer management overhead. Avoiding buffer management shaves a few instructions off the send and receive overheads and avoids the need for acknowledgement messages. More intangibly, virtual buffering removes the need for the user to reason about limited buffer resources and correctness, although it may still be desirable to limit buffer consumption for performance.

Virtual buffering improves memory performance by reducing the amount of physical buffer space required versus a system that pins its buffer pages in memory. The pool of physical page frames available on a node are effectively shared with other dynamic consumers of memory such as the demand paging system and the file cache. Section 5 presents evidence that the amount of buffer space required can be kept low.

Implementing virtual buffering in software keeps the size and complexity of the network interface low. 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 [13, 25, 29].

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 5.2.

We evaluate our implementation of the buffered path with a microbenchmark that causes many messages to be buffered. The overheads, including allocation of virtual memory on demand, are tabulated in Table 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.[2] 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.

[2] 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.

Item	Cycles
Minimum buffer-insert handler	180
Maximum handler (w/`vmalloc`)	3,162
Execute null handler from buffer	52

Table 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.

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. We call this principle *transparent access*, and it 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. We have identified 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.

The next section will explore the extent to which the unbuffered path is the common case of operation.

5 Experiments and Results

This section details experiments that show the performance of our implementation of UDM on a simulated FUGU system using real and synthetic applications. The results make three points. First, we show that applications naturally avoid buffered mode and that

App.	Description	Data set	Model	Measured	Cycles	Tot. msgs	T_{betw}	T_{hand}
Barnes	N-body simulation	2048 bodies, 3 iterations	CRL	3rd iter.	45.7M	107,849	3390	337
Water	Particle-in-cell	512 molecules, 3 iterations	CRL	3rd iter.	47.6M	36,303	10,500	419
LU	Blocked matrix decomp.	250x250 matrix, 10x10 blocks	CRL	all	13.4M	7,564	14,200	478
Barrier	10000 barriers	–	UDM	all	18.5M	240,177	615	149
Enum	Triangle puzzle	6 pegs/side	UDM	all	72.7M	610,148	953	320

Table 6: Application characteristics including runtime and messages sent for the measured part of each application running standalone on eight nodes. Programming models are native UDM messages and CRL, a software shared-memory system built on top of UDM.

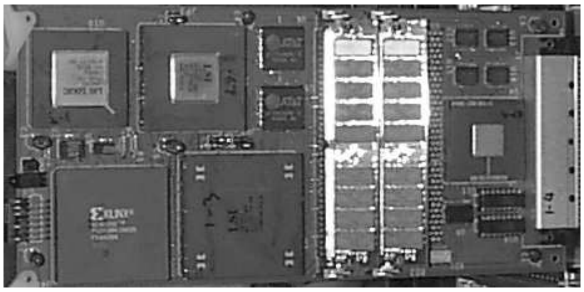

Figure 6: Prototype FUGU node board (25×12cm).

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, buffered versus direct delivery becomes a tradeoff against scheduling flexibility for applications that can tolerate buffering. Two-case delivery improves application performance by incurring buffering overhead only when the application is actually using the buffering to gain scheduling flexibility. Finally, we explore the limits to asynchronous messaging to determine the worst cases for our system. We show that although reasonable applications limit their demand for buffering, the buffering overhead should still be kept low because the throughput of the buffered case sets a limit for certain programming styles that synchronize infrequently.

Experimental Environment. The FUGU system is an experimental multiuser multiprocessor under construction. FUGU extends the MIT Alewife multiprocessor with the addition of a *User Communication Unit* (UCU) IC which implements the TLB, the GID stamp/check and a rudimentary second network. Our initial implementation of the UCU is based on a Xilinx 4025 FPGA.

A single node of a FUGU machine is depicted in Figure 6. The hardware currently runs most of the operating system and a few of our applications. The experimental platform used for the results in this section is a fast simulator used in advance of the prototype. The simulator models Sparcle instruction timing and the cache in the emulator: a single 64KB, unified cache with 16-byte lines and a 10-cycle miss penalty. The simulator design emphasizes speed over timing accuracy[3], but we believe the distortions introduced do not qualitatively affect our results. The current software system differs from the design presented in Section 4 in that the atomicity mechanisms are implemented in software (much like the RQ implementation on Alewife) instead of hardware. The GID stamp and check and the *divert-mode* bit are also currently implemented in software.

The FUGU operating system, Glaze, is a custom multiuser operating system based on the Aegis Exokernel [16]. The operating system supports multiprogramming, virtual memory, messages and user-level threads. Glaze implements the UDM model including virtual buffering used in response to GID mismatches and page faults[4], although message timeouts are currently fatal. The system scheduler, implemented as a user-level server, supports loose gang scheduling with synchronized clocks [19]. Overflow control is implemented partly cooperatively in a user library. Glaze, the scheduler and the benchmarks and synthetic applications described in the next section are all functional on the fast simulator.

The simulated system includes eight processors. The scheduler timeslice is set at 500,000 cycles. All numbers represent the average of three trials.

5.1 Application Performance

We evaluate the performance of our implementation by examining the effects of buffering in several applications tabulated in Table 6. Three applications, LU, Water and Barnes come from the SPLASH [31] suite and are slightly modified to make use of the CRL all-software shared-memory system [15]. CRL presents a message-passing load that is representative of coherence protocols such as Stache [24] and can be considered operating-system-like: many low-latency request-reply packets mixed with fewer larger data packets. A fourth application, `enum`, is a fine-grain, data-parallel application that exchanges numerous unacknowledged short messages and synchronizes only infrequently. At the other extreme, a synthetic application, `barrier`, included for illustration, consists entirely of barriers and thus synchronizes constantly.

Table 6 also shows the characteristics of the benchmarks running standalone on an eight-processor system. The table shows the number of cycles, the total number of messages, the average cycles between communication events (T_{betw}) and the average cycles spent per handler (T_{hand}) for each of the applications.

For our experiments we multiprogram each application, one at a time in separate runs, against a "null" application with schedules of varying quality. 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 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 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 makes the main point of the experiment: that the demand for buffering is relatively small and increases gracefully. Figure 7 plots the fraction of messages that take the buffered path ver-

[3]Compared to measurements on the hardware, the simulator reports cycle counts within +/-20%.

[4]Glaze does not support paging to disk, but does support faults to pages that are allocated and zero-filled on demand.

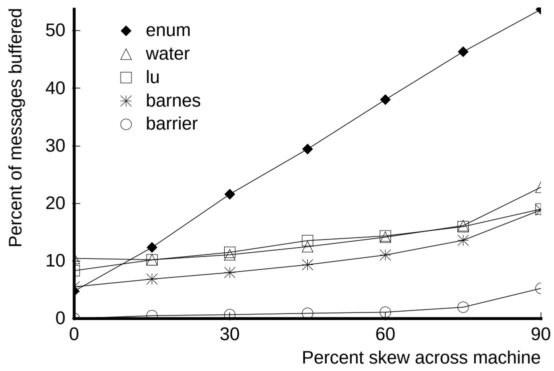

Figure 7: Percentage of messages traversing the buffered path for applications multiprogrammed with a null application versus decreasing schedule quality.

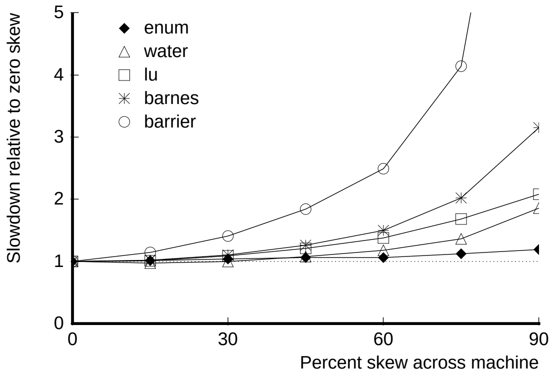

Figure 8: Relative runtimes of applications multiprogrammed with a null application versus decreasing schedule quality. Runtimes are normalized to the runtime of the application running alone.

sus 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 8 lists the relative runtime of each application normalized to the runtime of the application run with zero skew, which is within 1% of 2X 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.

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.

5.2 Buffering Behavior

If a node must start buffering, two factors help guarantee that the buffer will clear relatively quickly. The first is that any application that requires a reply after a message send inherently limits its own communication rate and limits the number of messages that can possibly be buffered at a given moment. The second is that an application that consumes messages faster than it produces them tends to clear the buffer even without synchronization. There remain applications that launch messages at very high rates for extended periods. At some point, however, an application written this way must simply be considered poorly behaved because it will perform poorly on any machine. On FUGU, an application that uses excessive buffering will eventually invoke the overflow control mechanism. We use a synthetic application to find the bounds on application behavior.

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 (T_{hand}) is fixed in our experiment at 290 cycles, including interrupt and kernel overhead. 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_{betw} cycles.

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 buffered mode periodically. Figure 9 presents the results, giving the percentage of messages buffered on the consumer node versus T_{betw}. There are two features to observe in the results. First, all versions of synth-N show a small percentage of messages buffered when $T_{\text{betw}} > (T_{\text{hand}} + T_{\text{buffering\_overhead}})$. In this region, the application is well-behaved by virtue of having a low 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 buffered mode only from the time buffered 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 10 demonstrates the importance of keeping the cost of the buffered path relatively small. In this experiment T_{betw} is held constant at 275 cycles, but we artificially added latency to the buffer handler. Again, synth-10 buffers only a small percentage of its messages because the benchmark's inter-

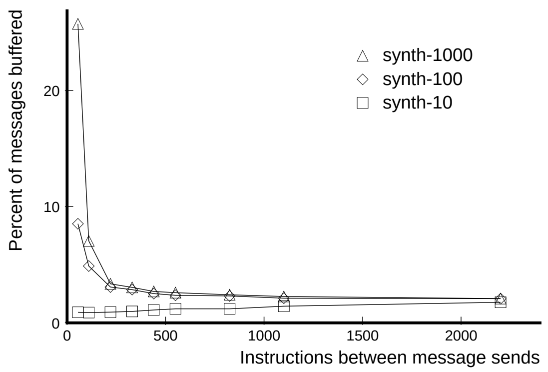

Figure 9: Percentage of messages buffered versus send interval with N messages (for synth-N) sent per synchronization point at 1% scheduler skew

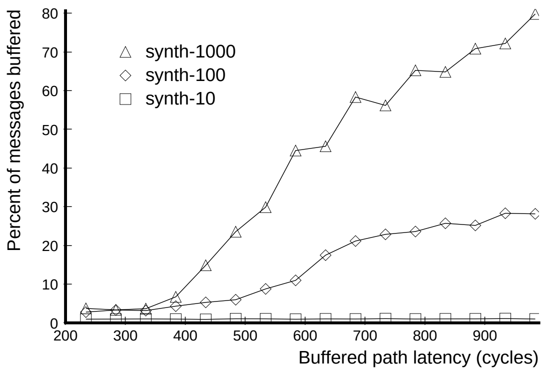

Figure 10: Percentage of messages buffered versus cost of the buffered path

nal 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 6, 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 under 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.

6 Conclusion

This paper describes how to efficiently protect and virtualize a tightly-coupled direct network interface. The key challenge is to provide native hardware performance in the face of multiprogramming and demand paging. The implementation meets this challenge using two-case delivery and virtual buffering. In the common, fast case, a process can read a message directly from the network interface. In cases where messages cannot be delivered immediately, the system transparently switches to buffering messages in virtual memory via software. Two-case delivery allows aggregate performance near the speed of the fast case. Virtual buffering allows automatic management of physical buffering resources for low physical memory requirements while providing effectively guaranteed delivery.

We have implemented the hardware portion of two-case delivery in a simulated FUGU multiprocessor. We also have fabricated a hardware emulator for FUGU. On top of FUGU we have implemented a multi-user operating system which provides the software parts of two-case delivery and virtual buffering. Results from experiments with microbenchmarks and applications running on top of Glaze and FUGU make three points:

- Message delivery in the fast case is fast: it is 87 cycles or about 60% more than unprotected message delivery (an estimated difference in runtime of 1-4% for the real applications in Table 6).
- Few physical memory pages are needed to support virtual buffering. Even if applications are artificially forced into virtual buffered mode, only relatively few physical pages are needed. Application synchronization characteristics tend to keep the real memory required for buffering low.
- The slow case, the virtual buffered case, needs to be reasonably fast. On FUGU and Glaze we have implemented a virtual buffering path which costs only 232 cycles. This is good enough to keep up with very high sustained message rates.

Acknowledgements

We would like to thank Robert Bedichek for the fast simulator, Jon Michelson for his work in the design and implementation of the UCU FPGA and Kirk Johnson for the CRL and `enum` benchmarks. The FUGU project has been funded in part by NSF grant # MIP-9504399, 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.

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, MIT, Computation Structures Group, Cambridge, MA, 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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Kenneth M. Mackenzie. *The* FUGU *Scalable Workstation: Architecture and Performance*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1998.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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*, April 1994.

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

[29] 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.

[30] 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.

[31] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[32] 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.

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

[34] 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.

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