UDM: User Direct Messaging for General-Purpose Multiprocessing

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

March 8, 1996

Abstract

User Direct Messaging (UDM) allows user-level, processor-to-processor messaging to coexist with general multiprogramming and virtual memory. Direct messaging, where processors launch and receive messages in tens of cycles directly via network interface FIFOs as opposed to indirectly via memory, offers high message bandwidth and low delivery latency by avoiding memory delay and buffer management overhead. However, user-level direct messaging implementations to date are limited in that they operate only in single-user machines or with strict gang scheduling. In this paper, we develop a messaging approach for protected, direct delivery with a single, unified user interface but with an underlying implementation that provides two delivery cases: a fast, common case corresponding to direct user access to hardware queues and a second case using virtual buffering that is invoked transparently when required by the demands of multiprogramming, virtual memory or user intransigence.

The paper lays out a simple, efficient messaging model for user direct messaging that allows both user interrupts and user polling by explicitly incorporating atomicity. The paper then identifies two mechanisms that enable the model to map to a fast, hardware path: a *revocable interrupt disable* mechanism in hardware permits the user to block the network in a limited way and an *overflow control* scheme used in the virtual buffer case allows all buffer management overhead to be avoided in the fast path. Experiments with real and synthetic applications on an existing, single-user machine, Alewife, and a new, simulated, multi-user machine, FUGU, show that the cost of the fast case is within a few cycles of the cost of unprotected, kernel messaging and indicate that the fast case can indeed be expected to be the common case under ordinary conditions.

Keywords: Network Interface, Multiprocessing, Multiprogramming

*This research has been funded in part by NSF grant # MIP-9012773, in part by ARPA contract # N00014-94-1-0985, and in part by a NSF Presidential Young Investigator Award.

1 Introduction

The maturation of complex software design has lead away from single-threaded applications running on monolithic operating-systems toward multi-threaded applications interacting with micro-kernels. In fact, the "client-server paradigm" has been elevated to the status of a major organizational principle. Two salient components of this paradigm are protection domains and light-weight threads. The use of multiple protection domains greatly enhances fault containment and debugability – even within an operating system. Light-weight threads, on the other hand, greatly simplify the construction of applications which deal with complex, interlocking events, such as modern windowing systems. Closely related to these developments is the drive to reduce the cost of major operating systems services, from interprocessor communication to address translation and file access. One of the most successful techniques for reducing overhead is specialization: providing customized interfaces which support only those features which are actually needed by a given application. This approach recognizes that compilers and application writers are in the best position to evaluate the needs of individual applications; hence, the current trend toward exporting hardware facilities directly to user level [2, 7].

While operating systems research has been addressing the construction of complex systems, multiprocessor research has focused on raw performance, primarily because the users of parallel computers have been concerned more with performance than with usability. Multiprocessor research has lead to low-overhead, fine-grain communication mechanisms, including efficient shared-memory and message-passing interfaces, fast asynchronous interrupts, and rapid context-switching. However, combining communication performance with protection and other usability features, such as virtual memory, taken for granted in modern operating systems has not been straightforward. Memory-based communication such as shared-memory or bulk, memory-to-memory message passing can use the same memory protection mechanisms as are used in uniprocessors [3, 22]. On the other hand, systems supporting fine-grain message passing are currently either single-user machines, at best resorting to hard partitioning or strict gang scheduling to permit multiprogramming [17, 6, 1, 11] or use alternate techniques that generally add restrictions or overhead (see related work in Section 6).

This paper develops a model of messaging called *User Direct Messaging* (UDM), which allows the application of the techniques of modern operating systems to multiprocessors without sacrificing the efficiency of single-user multiprocessor communication mechanisms, *i.e.,* it combines support for the requirements of client-server decomposition and multiple protection domains with direct, user-level access to hardware. Our goals in designing UDM are threefold. First, the programmer should be able to maintain a natural view of the network as a single-user, dedicated resource. Second, an application must operate correctly despite scheduling uncertainty introduced by multiprogramming and virtual memory. Finally, the application should achieve essentially the same performance as on a single-user machine provided the system scheduler succeeds at *co-scheduling* (gang scheduling, but only loosely and on demand [14, 18]) the processes in the application.

UDM consists of a user-visible messaging model supported by a transparently two-level implementation with a fast case corresponding to direct access to hardware queues and a buffered case invoked when required.[1] UDM has at its root a simple, *single-user* model of messaging: the user appears to have exclusive and direct access to the network hardware. The model is similar to Active Messages [21] but explicitly defines and recognizes *atomic sections* in which the user is given (apparent) privilege to disable message arrival interrupts for polling and interrupt handler management. The result is fast,

[1] We focus in this paper on the UDM communication mechanism; our system also supports complementary communication mechanisms including DMA for bulk transfer and hardware-synthesized messages for accelerating shared memory.

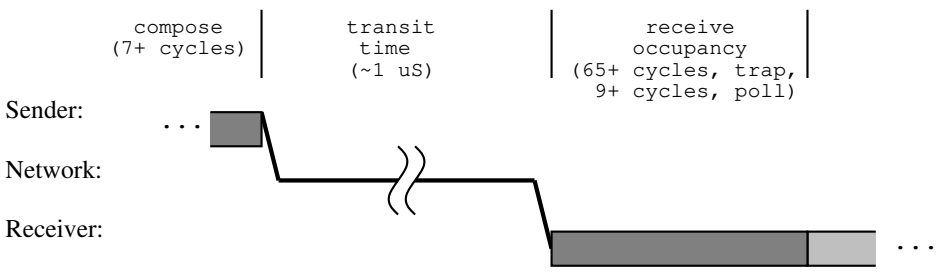


Figure 1: Direct message timing.

low-overhead, user-level messaging that smoothly integrates interrupts and polling.

The implementation utilizes a combination of hardware and software mechanisms. In the fast case, the model is matched exactly by a network interface supporting direct messages and a hardware *revocable interrupt disable* mechanism that gives the user-level code limited power to disable interrupts (blocking the network) while user handlers run. Direct messaging is an established technique for particularly low-overhead message passing in which the network interface is integrated closely with the processor and messages are logically and physically passed from processor to processor with no traversal of the memory system [17, 6, 11, 8, 1]. Low overhead and latency are achieved by avoiding the memory system, so that message overheads scale with processor performance rather than with memory performance. Figure 1 represents the one-way message latency and overheads predicted for our prototype system, FUGU, operating at 20MHz. By avoiding memory, the overhead cycle counts can be expected to roughly predict performance for modern and future systems as well: assuming the handler is active enough to be found in the cache, the receive handler can complete in less time than a cache fill from DRAM on a modern processor. The revocable interrupt disable mechanism allows the application to fully participate in low level flow control and scheduling decisions, in the fast case.

The fast case is protected by hardware against exceptional conditions such as incoming messages that arrive for an application that is not currently running. In response to these exceptions, the operating system transparently switches message delivery to a virtual buffering system. The virtual buffering system stores messages in application virtual memory and includes an *overflow control* mechanism that allows message delivery to be guaranteed, subject only to the constraint of the size of virtual memory. The operating system "handler" that buffers messages in memory is roughly as fast as a minimal user handler, so the (short-term) bandwidth of messages into the node changes only slightly although the effective latency of a message increases significantly. A co-scheduling system scheduler assures that the fast case is in fact the common case so that buffering is invoked only in response to scheduling transients or other uncommon events.

UDM represents a synthesis of techniques, many of which are not new. The unique aspect of the UDM model and implementation is that it provides the user with direct user-level access to network hardware and interrupts while remaining amenable to transparent multiplexing between different protection domains. Specifically, we make three contributions: (1) The UDM model codifies atomicity semantics as part of user messaging. (2) The revocable interrupt disable hardware mechanism permits user handlers to interact with the network's low-level flow control in a protected way and is key to fast message processing. (3) The virtual buffering and overflow control strategy provides unlimited buffering without impacting the performance of the common case path.

The remainder of the paper is organized as follows. Section 2 presents the programmer's view

of the UDM messaging model. Section 3 lays out the architecture for a protected implementation of the model for a multiprogrammed multiprocessor. Section 4 describes the implementation of our prototype. Section 5 describes the results of experiments to validate the design. Section 6 puts our work in perspective by describing related work and, finally, Section 7 concludes.

2 UDM Model

This section describes the UDM programmer's model in abstract terms. The interface may be used directly in applications or indirectly via a library or compiler to provide higher level or specialized communication primitives. UDM allows the programmer to view the network as a private resource. Therefore, the UDM model may be discussed independently of its implementation, as if the network were dedicated to a single application. Section 3 describes the hardware implementation and the mechanisms devoted to maintaining the appearance of a dedicated network in a multiuser environment.

There are two points in this section. 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, UDM provides for explicit control over interrupts in user code allowing the integration of both polling and interrupts 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 [21]. 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 [9]. Message injection can thus be viewed in the following fashion:

```
send(header, handler, word0, word1, ...)
```

If resource contention prevents the network from accepting a given message, the corresponding `send` operation *blocks* until successful. Alternatively, blocking can be avoided by using a conditional, non-blocking version of `send`, namely `sendc`:

```
sendc(header, handler, word0,word1,...) ⇒ true|false
```

`Sendc` returns a boolean condition which indicates whether or not the message has been successfully injected into the network; it is up to the user to retry the send operation if it is not injected successfully. Once a message has been injected into the network, the UDM model guarantees that it will be eventually 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 which reads the contents of the message and frees it from the network:

```
receive() ⇒ (header, handler, word0, word1, ...)
```

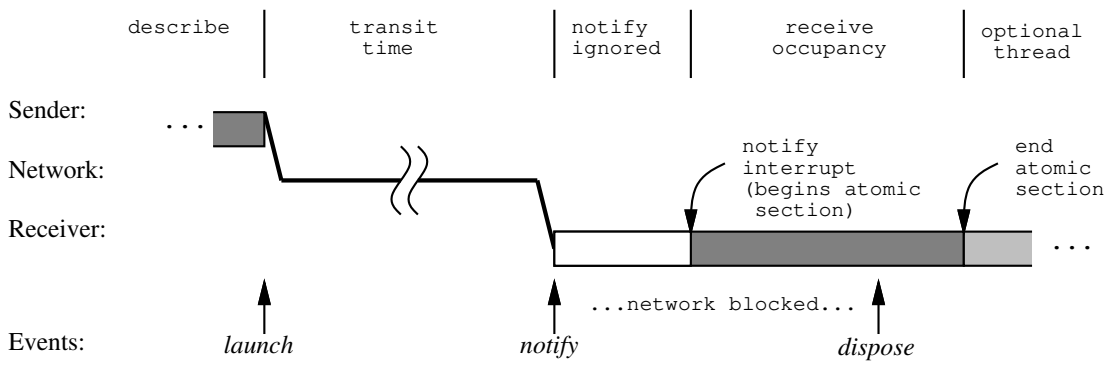


Figure 2: Message time line for interrupt delivery on the fast path.

Implicit in this syntax is the fact 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 a `receive` operation will succeed. It is an error to attempt a `receive` operation when no message is available.

In addition to the `receive` operation stated above, UDM provides a mechanism for quick examination of the next message without actually extracting it from the network:

```
peek() ⇒ (header, handler, word0, word1, ...)
```

As above, it is an error to attempt a `peek` operation when no message is available.

Execution Model. UDM assumes an execution model in which one or more *threads* run on each processor. A single interrupt disable bit determines whether or not interrupts are used for notification of message arrival. Periods of execution in which message 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 `receive` 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 handler begins execution with message interrupts disabled, at the handler address specified in the message. A handler is required to extract one or more messages 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 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. Figure 2 illustrates the timing of a message.

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 ordinary, if ad hoc, in kernel-level device drivers. Providing this control at user level allows user code (in libraries or specialized by the programmer or compiler) to interact with the network interface with the same efficiency and flexibility as kernel code. The interaction of atomic sections and protection in a multi-user environment will be discussed in detail in the next section.

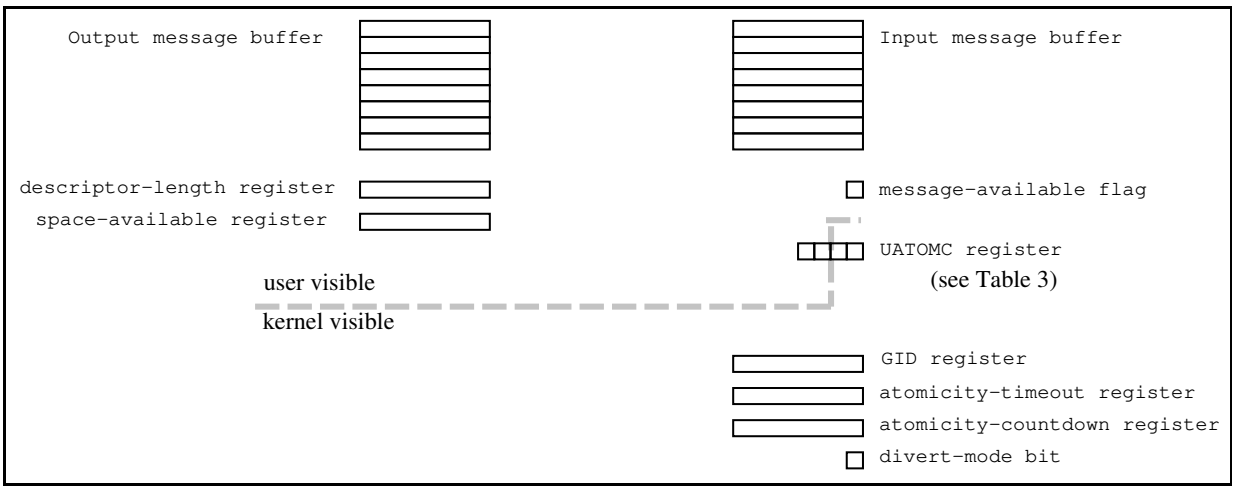


Figure 3: UDM network interface registers.

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 *descriptor-length* **:=** 0
`dispose`	**If** *message-available* not set **then** cause a *bad-dispose* trap, **else** delete current incoming message.
`beginatom(MASK)`	*UATOMC* **:=** (*UATOMC* $\vee$ `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** *UATOMC* **:=** (*UATOMC* $\wedge$ ($\sim$`MASK`))

Table 1: UDM operations

3 UDM System Design

This section details the design of the UDM system. We give an ISA-level description of the memory-mapped network interface and then describe its use in the common, fast case. The fast case includes hardware protection to support multiprogramming. The central feature of the fast path is the revocable interrupt disable mechanism which permits user interrupts and direct polling. Finally, we describe the alternate, buffered delivery path, how it provides semantics identical to the fast path mode and how the overflow control mechanism is used to limit buffer usage.

3.1 Network Interface

The UDM 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 our system 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

Interrupt/Trap	Event Signaled
message-available	User interrupt: raised when a message is available for reading
mismatch-available	Interrupt: message available with mismatched GID
atomicity-timeout	Interrupt: atomic section timer expired
atomicity-extend	Trap: optionally triggered by end of atomic section
dispose-extend	Trap: optionally triggered by `dispose`
dispose-failure	Trap: triggered when user fails to free message
bad-dispose	Trap: attempt to execute `dispose` with no pending message.
protection-violation	Trap: user mode access to kernel interface registers *or* user mode attempt to send a kernel message

Table 2: Interrupts and traps

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 3.3.

Send and Receive. The `send` operation of the abstract model is decomposed into a two-phase process of *describe* and *launch*, as used in [9]. 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. 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 `send` 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 `receive` 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 `receive` 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 and the details of the interaction of user code with user interrupts is the topic of Section 3.2 on the revocable interrupt disable mechanism.

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 fully

in Section 3.2).

3. A reserved, second network exists for occasional use by the operating system in situations otherwise subject to deadlock (see Section 3.3).

The GID labels a group of processes 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.

UDM 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.

3.2 Revocable Interrupt Disable

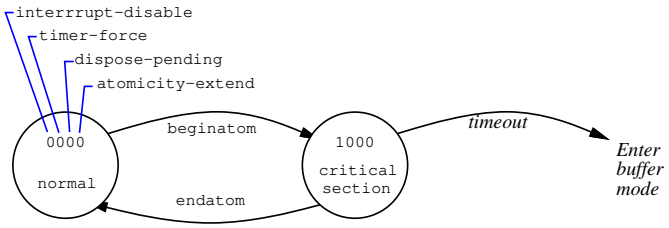
UDM's revocable interrupt disable hardware gives applications the ability to lock out network interrupts temporarily when receiving in the fast case. This fine control over the network and the processor enables efficient message reception by allowing the user to receive messages directly from the network interface. From the user's perspective, the hardware provides the user three things:

1. Fast, user-level *message-available* interrupts.
2. Critical sections atomic with respect to *message-available* interrupts.
3. User-level polling directly from the network interface.

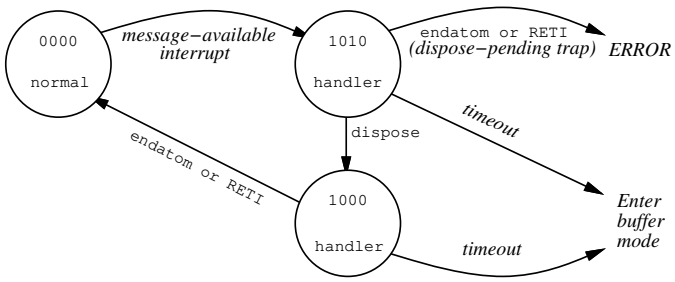
However, for protection, the kernel must retain the ability to recover control when required. Thus, the hardware must also address the following issues:

- For the network to make forward progress, the time from the arrival of a message at a node to the `dispose` operation (*notify* to *dispose* in Figure 2) must be bounded.
- If the *message-available* interrupt preempts high-priority code, the atomicity mechanism must assure that the processor correctly reschedules the high-priority code at the end of the atomic section and that the period of the atomic section ("receive occupancy" in Figure 2) is bounded.
- The atomicity mechanism must operate transparently whether messages are arriving via the fast path or via the buffered path.

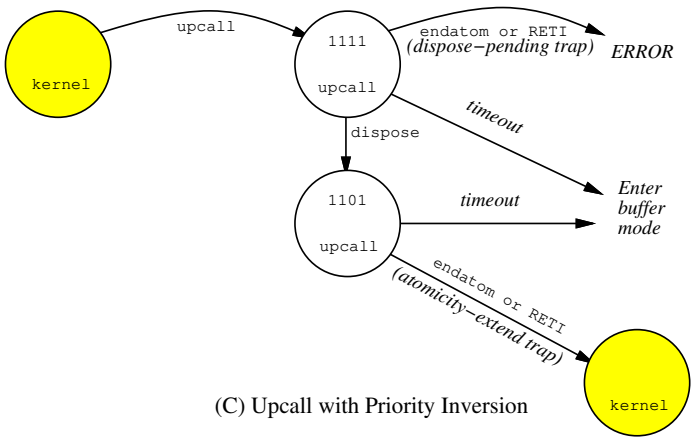
Note that the "atomicity" provided by the revocable interrupt disable mechanism in hardware is identical to the atomicity in the user's model (from Section 2) only when the fast path is in use. On the buffered path, atomicity is provided to the user by software in the way the buffered queue and thread scheduling are handled. The system may switch a message from the fast to buffered path transparently at any time.



(A) Critical Section / Polling



(B) User–Level Interrupt



(C) Upcall with Priority Inversion

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

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

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

Table 3: Detail of atomicity control bits in the *UATOMC* register (Figure 2).

Control over user-level interrupts is implemented by a dedicated atomicity timer, four atomicity control bits in the *UATOMC* register, and the `beginatom` and `endatom` operations. Table 3 details the control bits in the *UATOMC* register. Two of the bits are only modifiable 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, *atomicity-countdown*, and a preset value, *atomicity-timeout*. While the timer is *disabled*, *atomicity-countdown* is preset to the *atomicity-timeout* value. When the timer is *enabled*, *atomicity-countdown* decrements for each *user* cycle, flagging an *atomicity-timeout* interrupt if it reaches zero. The atomicity timer is enabled during atomic sections by the user *UATOMC* bits, as described in Table 3. The use of the atomicity controls is best described by example. Figure 4 illustrates the progression of states of the *UATOMC* register for each of the following cases:

Critical Section. A critical section (Figure 4A) for a thread can be constructed by wrapping it with an `beginatom` instruction that sets *interrupt-disable* and optionally *timer-force* bits in the *UATOMC* and an `endatom` instruction that resets both bits. The overhead of the critical section is thus two instructions. The use of *timer-force* is optional. If *timer-force* is set, then the timeout timer bounds the period of the critical section exactly, which is useful for debugging. If *timer-force* is clear, the timeout timer bounds only the time from the arrival of a message (if any) to subsequent message handling.

Polling. Polling is constructed as an "extended" critical section (Figure 4A) that periodically presets the timer. Again, the use of *timer-force* is optional. If clear, the timer is preset automatically every time a message is handled, and the overhead of polling is limited to a load and test of the *message-available* flag per poll. The timeout timer bounds the time from the arrival of a message to subsequent message handling. For debugging purposes, it is useful to bound the period of the polling loop, in which case *timer-force* is set. A poll is then coded by briefly clearing *timer-force* while checking *message-available*. If no message is at the input, clearing *timer-force* presets the timeout timer. Otherwise, the `dispose` in the message handler presets the timeout timer. This more conservative polling loop requires two instructions of extra overhead beyond the load and test of *message-available*.

User-Level Interrupt Handler. An application interrupt handler (Figure 4B) is implemented by setting *interrupt-disable* and also *dispose-pending* in *UATOMC* before transferring control to the user handler. *Dispose-pending* records that the application must process a message before exiting the handler's atomic section or returning to the kernel from the interrupt (RETI). *Dispose-pending* is implicitly cleared when the application handler executes the `dispose` instruction after reading the message from the input buffer. The timeout timer bounds the time from the arrival of the message to the `dispose` operation (if *timer-force* is clear) or, optionally, to the end of the atomic section.

Our system supports user interrupts only by having a kernel handler quickly convert a kernel interrupt into an upcall to user code and adding a system call to perform a return from interrupt. Thus, the manipulations to *UATOMC* are added in software to the kernel handlers. However, a processor with user interrupts supported in hardware could manipulate the *UATOMC* bits as part of the user interrupt mechanism, leading to an implementation that could be considered the ultimate goal of UDM: providing protected messaging while in the common case removing all software between the user and the network at both the source and destination of messages.

User Handler with Priority Inversion. The revocable interrupt disable mechanism allows the kernel to synthesize a protected *upcall* to a user handler (Figure 4C). The kernel implements the upcall by setting the *atomicity-extend* and *timer-force* bits in the *UATOMC* register before transferring control to user code. *Atomicity-extend* causes any `endatom` operation in the handler to trap to the *atomicity-extend* trap, unconditionally returning control to the kernel at the end of the handler's atomic section. *Timer-force* causes the atomic section to be timed regardless of the presence of a message, bounding the time that the user code can lock out kernel code.

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. Timeouts cause the message, if any, to be copied to buffer storage and the handler to be rescheduled at a later time. A timeout of zero corresponds to a system that always invokes buffering. An infinite timeout corresponds to no protection. The timeout value may be varied by the operating system for tuning or to gather statistics.

3.3 Virtual Buffering and Overflow Control

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 the virtual buffering are to provide:

1. Identical semantics to the fast case and a transparent transition between cases.
2. Graceful degradation of performance as demand for buffering increases.
3. Guaranteed delivery, enabling the fast case to avoid all buffer management overhead.

We address the objectives as follows. First, hardware and software mechanisms and software conventions provide transparency. Second, the software buffer itself is *virtualized,* allowing essentially unlimited buffering while avoiding dedicating physical memory to what is presumed to be an infrequent case. Finally, the overflow control mechanism makes guaranteed delivery and unlimited virtual buffering practical by providing feedback from the buffering system to the system scheduler.

Virtual Buffering Mode. Buffered delivery is a *mode* entered when fast path delivery is not possible, *e.g.,* because a mismatched message arrived or because of a timeout or page fault in an atomic section. Each of these situations causes a kernel interrupt or trap that in turn switches to buffered mode. Buffered mode is a per-processor, per-application state. 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. An application on a processor remains in buffered delivery mode until the last buffered message is processed and the software can revert to allowing user messages to be received directly from the network interface. Buffered mode is made transparent to the user by hardware features of the network interface enabled by the *divert-mode* bit, by operating system software for buffer management and thread scheduling, and by software convention in the use of the 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 loads

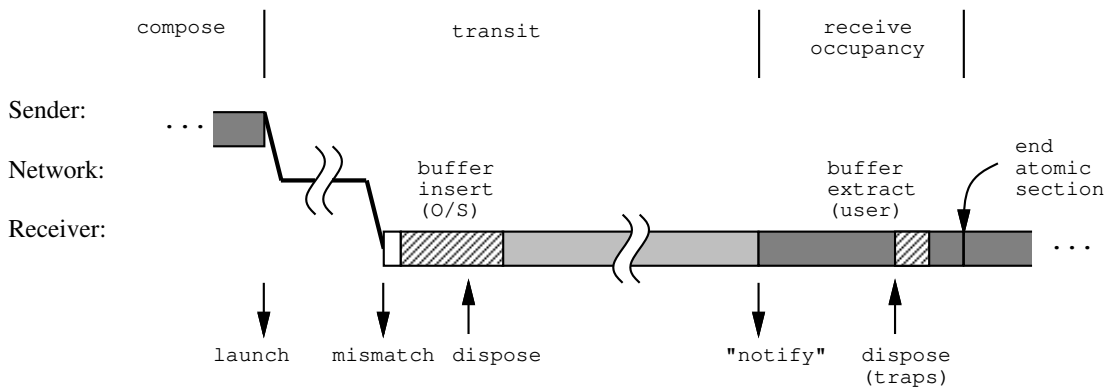


Figure 5: Message timeline for the buffered path.

new messages into the software buffer of the application indicated by the GID in the message header.[2] In addition, *divert-mode* set causes the user-mode `dispose` instruction to take the *dispose-extend* trap. The *dispose-extend* trap handler 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.[3]

The buffered delivery mode presents the user with the same atomicity semantics as the fast path hardware by a combination of buffer management and scheduling. 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. If the application was polling for messages at the time buffering was invoked, the polling thread continues to operate but reads messages from the software buffer instead of from the network interface. If instead *message-available* interrupts were enabled, a message cleanup loop is scheduled as the background user thread on the processor and runs the atomic section of each handler (using *atomicity-extend* to limit execution to the atomic section) until the software buffer is empty.

Given the above hardware and software features, the buffered receive interface can be made completely transparent to application code by employing a software convention of using a known base register to point to the hardware's input message buffer. 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. Handler code is thus identical for both fast and buffered cases, an important feature from a software engineering perspective. This convention allows an incoming message to be transparently moved from the hardware to the software queue even if the message handler is in the middle of reading the message.

Virtual Buffering. 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. The solution in UDM is to buffer messages in virtual memory, effectively sharing the node's pool of free page frames with other dynamic consumers such as demand paging and file caching. The best policy for allocating page frames is an open question; we currently treat physical memory as a unified cache.

[2]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.

[3]There are other possible recovery policies. The strategy we describe preserves message ordering (because the network used happens to preserve ordering). A variant we intend to pursue is the idea of recovering from buffering by interleaving the handling of buffered messages with that of new, directly-handled, messages.

Virtual buffering reduces the pages that must be pinned compared to physical buffering. Only the head page, if any, of the software queue for each application that may receive messages at a node must be resident. The interrupt/trap code that inserts a message into the software buffer either adds the message to the existing, partly filled, head page or extends the software buffer by quickly allocating a fresh page. The buffer insertion code only stalls when there are absolutely no page frames available on the node. This unlikely but not impossible situation is addressed by a separate mechanism, described next.

Overflow Control. Overflow control deals with limited buffer resources. UDM guarantees message delivery so all messages that cannot be delivered directly must be buffered in memory. Buffering is expected to be rare given effective co-scheduling and, further, the buffering system has access to all free page frames on the node, so the buffering system is unlikely to run out of physical memory. However, we provide an escape mechanism from buffer overflow because running out of memory is fatal and because the memory demands are unpredictable: (1) the coscheduler takes time to repond to changes in traffic rates, (2) it is difficult to reason about (and undesirable to limit) exactly how many outstanding messages an application may generate, and (3) even if applications obeyed a strict limit, virtual buffering implies that the exact number of free page frames available is unpredictable.

UDM relies on a system scheduler that ordinarily co-schedules applications based on their communication needs with the goal of keeping the system in the fast delivery mode.[4] The overflow control idea is that when a software message buffer reaches a high-water mark (or, alternately, the memory system's free page frame list reaches a low-water mark), the buffering system invokes the system scheduler *immediately* and *globally* to deschedule the offending job, effectively performing flow control on the source of messages that are filling the buffer. After memory has been freed up (by paging), the job is then globally restarted, presumably with more attention paid to coscheduling, and allowed to proceed. An ordinary application will eventually empty the buffered messages and return to the fast case. An unbalanced or faulty application may continue to consume buffer space, but the network-overuse problem has been converted into a (virtual) memory consumption problem. Experiments in Section 5 explore what it means for an application to be well-behaved.

If the memory system has completely run out of free pages, as opposed to merely reaching a low-water mark, the main network is deadlocked until the memory can be freed. To avoid deadlock, we use the second network, reserved to the system, for flow-control messages and possibly for paging in deadlock situations. The performance demands of the overflow control mechanism on the second network are slight, however. Our proposal includes only a rudimentary second network.

4 Implementation

The UDM model provides single-user messaging semantics that can be used in either single- or multiuser systems. We have implemented UDM on two systems: a single-user, physically addressed machine, Alewife, running on real hardware, and a multiprogrammed, virtually addressed machine, FUGU, currently running on a simulator with hardware under construction. The hardware base of the two machines is similar, allowing us to calibrate the FUGU simulator with the Alewife hardware. This section describes the two platforms and their current limitations.

Alewife. The Alewife system is a single-user, physically addressed multiprocessor consisting of single-processor nodes distributed on a mesh network. Each Alewife node consists of a Sparcle (SPARC V7

[4] Statistics from the buffering system may contribute information useful for coscheduling even under ordinary conditions.

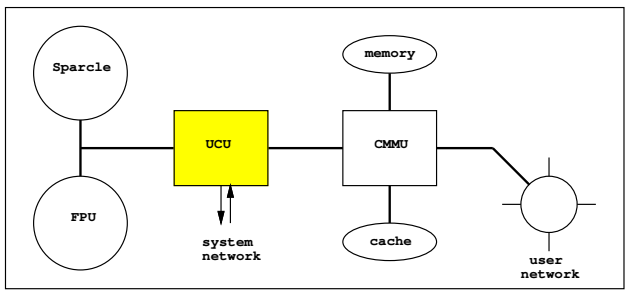


Figure 6: System block diagram for FUGU (with the UCU) or Alewife (without the UCU).

derived) processor, 64K bytes of direct-mapped cache, 8 Mbytes of DRAM main memory, a SPARC floating-point coprocessor and an Elko-series mesh routing chip (EMRC) from Caltech that forms a part of the direct, wormhole routed network. A single-chip *Communications and Memory Management Unit* (CMMU), implements cache control, memory control and the processor's interface to the network. Alewife currently operates at 20MHz with an upgrade to 33MHz planned.

The Alewife hardware and software provides a single-user version of UDM where the high level model is backed by a fast case implementation only. Multiprogramming and page faults do not occur and message handler timeouts terminate the user program, although the timeout period is adjustable for debugging. The Alewife hardware includes memory-mapped send and receive queues and single instruction `launch` and `dispose` operations. The Alewife CMMU also serves as the interrupt controller and, in a forthcoming revision, will implement the revocable interrupt disable mechanism. The current Alewife system emulates the atomicity system using the system timer and a reserved flags register in the processor at a cost of approximately 30 extra cycles per interrupt handler [4].

The Alewife software system provides a particularly aggressive use of extremely lightweight threads: interrupt handlers execute as general threads that can block and that possess their own stacks. To achieve this speed, we take advantage of the existence of two or more independent register sets. Such support is an integral part of the SPARC architecture and appears in other architectures as well.

FUGU. The FUGU system is an experimental multiuser multiprocessor under construction. FUGU extends Alewife with the addition of a *User Communication Unit* (UCU) IC which implements the TLB, the GID stamp/check and a rudimentary second network. This UCU IC has been designed and is currently in layout. The initial implementation will be based on a Chip Express ASIC. Our experimental platform is a fast but not timing-accurate simulator used in advance of the prototype. The current system differs from the design presented in the paper in that the atomicity mechanisms are still implemented in software (much like Alewife) instead of simulated hardware. The GID stamp and check are also currently implemented in software. We account for these differences in the results by explicitly counting the costs in cycles of the software emulation.

The FUGU operating system, Glaze, is a custom multiuser operating system based on the Aegis Exokernel [7] and is under active development. 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,[5] although message timeouts are currently fatal. The system scheduler, implemented as a user-level server, is only rudimentary at this time but supports both gang scheduling and a simple form of co-scheduling by synchronizing processes

[5]Glaze does not yet support paging to disk, but does support regions that are zero-filled on demand

Item	Alewife/FUGU kernel mode (cycles)	Alewife, Rev1 user mode (cycles)	FUGU user mode (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
Interrupt entry	16	16	10
Timer setup	–	13	1
Dispatch	10	15	15
subtotal:	32	50	32
Null handler (w/`dispose`)	3	5	3
Timer cleanup	–	19	1
Interrupt exit	17	21	17
interrupt total:	**52**	**95**	**65**
Message Receive (polling)			
Poll	3		3
Dispatch	5		5
Null handler (w/`dispose`)	1		1
polling total:	**9**		**9**

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 Alewife numbers include overhead to emulate the atomicity mechanism on the first silicon CMMU.

periodically. Overflow control is not yet implemented. Glaze, the scheduler and the synthetic and `Tpuz` applications described in the next section are all functional on the fast simulator.

5 Experiments and Results

This section details experiments that show the performance of UDM on the FUGU system and some comparisons to performance on the Alewife system. The results are based on FUGU simulations as well as runs on the existing Alewife hardware which uses largely the same set of hardware components as FUGU. The experiments make three points. First, the protected, user-level "fast path" provides communication between users at essentially the same speed as unprotected, kernel-level messages running on raw hardware while the buffered path has an overhead of approximately 3.5 times the fast path. Second, we show that it is reasonable to expect well-behaved programs to stay out of buffering mode, but that it is important to keep the buffering overhead low. Finally, we exhibit some preliminary results from multiprogramming that demonstrate the flexibility of the system: strict gang scheduling is not required.

5.1 Basic Costs

Fast Path. Table 4 details the cost of sending and receiving messages in FUGU at user level, in Alewife at

Item	Cycles
Minimum buffer-insert handler	163
Maximum handler (w/`vmalloc`)	2068
Execute null handler from buffer	71

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.

user level and at kernel level in either machine. The Alewife cycles breakdowns come from [4], verified by microbenchmarks run on the Alewife hardware. The kernel and FUGU cycle counts are made from simulator traces. The send cost is for a blocking `send` 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 FUGU simulation, but we predict the performance of the FUGU prototype by eliminating the appropriate categories. The atomicity mechanism will be implemented in a second revision of the CMMU part and the the GID stamp and check are implemented in the UCU. The result is that the overhead of the fast path for user-to-user communication in FUGU is within a few cycles of the overhead for unprotected, kernel-to-kernel communication.

Buffered Path. In the common case messages will be delivered via the fast path. If, however, a message arrives for a process that is not immediately ready to receive it, Glaze will buffer the message in the virtual buffer and then deliver the message to the process at the next opportunity. The buffered path introduces two components of overhead. First, there is an extra copy operation: a Glaze handler must copy the message from the network interface to memory. Second, the user handler must 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 below in Section 5.2

We evaluate the Glaze 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 234 (= 163 + 71) cycles, or about 3.5 times the fast path overhead of 65 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.[6] 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 (2068 cycles), but occur so rarely as to be negligible in our

[6]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.

simulations.

5.2 Buffering Behavior

We expect applications to observe buffering overhead only rarely because buffer 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 studied the incidence of buffering with a synthetic application.

If a node must start buffering, several factors help guarantee that the buffer will clear relatively quickly. 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 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 percentage 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. 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 on two processors with various amounts of synchronization. At the consumer node, each incoming message from the producer invokes a handler that stalls for a short period, and then sends a reply message. The handler time is fixed at T_{handler} cycles, not including messaging overhead. The producer iteratively generates groups of N messages, 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 T_{handler} fixed at 750 cycles and N set to 10, 100 and 1000 messages. We artificially induce the application to switch to buffering mode periodically by employing a scheduling trick: we multiprogram synth-N with a "null" application (one that runs an infinite loop). The scheduler gang-schedules the two applications, but artificially introduces a skew of 2000 cycles between the context switch points on the two processors. Thus, at the beginning of each (.5Mcycle) timeslice, there is a 2000 cycle window during which a synth-N producer message arriving at the consumer node will generate a *mismatch-available* interrupt, forcing the consumer node of synth-N into buffering mode.

Figure 7 presents the results, giving the fraction 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 percentage 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.

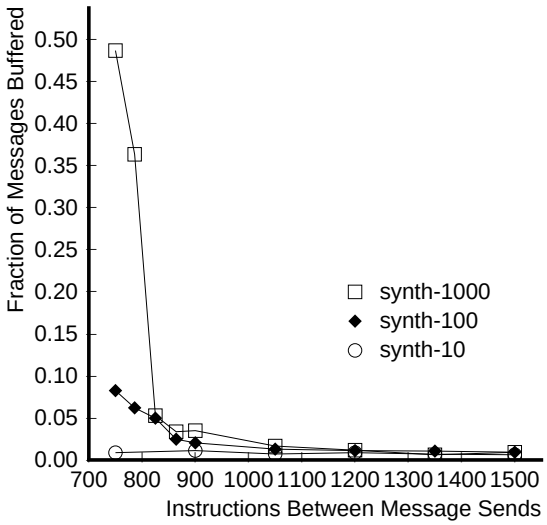


Figure 7: Percentage of messages buffered at the consumer versus send interval for a fixed 750 instruction handler and N messages (for synth-N) sent per synchronization point

Workload	Batch Scheduled		Gang Scheduled		Co-Scheduled	
	Instrs.	Total Messages	Normal Instrs.	Msgs. Buffered	Normal Instrs.	Msgs. Buffered
`Tpuz`	120.6M	513.5K				
`Tpuz` vs. null	239.6M	513.5K / 0	1.009	2 / 0	1.008	12.1K / 0
`Tpuz` vs. `Tpuz`	237.4M	513.5K / 513.5K	1.019	172 / 191	1.059	147.9K / 144.3K
`Tpuz` vs. synth-100	235.6M	513.5K / 280.0K	1.010	63 / 17	1.189	202.3K / 62.6K
`Tpuz` vs. synth-10	236.6M	513.5K / 280.0K	1.010	61 / 20	1.377	167.4K / 6.5K

Table 6: Instructions executed and messages queued in FUGU for several workloads and three scheduling strategies on four processors. The timeslice interval is 0.5M instructions.

5.3 Multiprogramming

Finally, we illustrate the flexibility of the system by running an application multiprogrammed as part of workloads under several scheduling strategies. The application, `Tpuz`, performs an exhaustive search of board positions in the "triangle puzzle" [12] — a simple game. The application operates in stages separated by barriers, where at each stage processors exchange many unacknowledged messages with one another. The messaging in `Tpuz` can be characterized by $T_{\text{handler}} \simeq 170$ instructions and $T_{\text{interhandler}} \simeq 950$ instructions. We run `Tpuz` alone as a baseline and then in conjunction with null (which in this case runs a loop for a fixed number of iterations), with itself and with 4-processor variants of synth-N ($T_{\text{handler}} = 500$ and $T_{\text{interhandler}} = 1000$). The scheduling strategies are batch scheduling, gang scheduling and rudimentary co-scheduling. Table 6 summarizes the results for for each workload. The columns for the batch run give the total run time (in simulated instructions) of the workload and the total number of messages sent for each job in the workload. The other columns give the run times normalized to the batch case and the numbers of messages buffered for each job.

The gang scheduler operates by giving the machine to an application job for fixed-size timeslices and ignoring any `yield` calls made by the application. The scheduler does not use barriers but rather relies on the ability of the hardware to make timeslice interrupts occur roughly simultaneously on all nodes. Gang scheduling adds a small cost (1-2%) for the overhead of the scheduler. There are also a small number of messages buffered as a result of imperfections in the schedule. It is a feature of UDM that it can gracefully tolerate such imperfections.

The results for the rudimentary co-scheduler demonstrate the flexibility of the system to support multiple scheduling policies. Our simple co-scheduler operates by synchronizing jobs periodically (every 15th timeslice), but otherwise allows the processors to switch processes independently at the end of each timeslice or in response to `yield` calls. Both `Tpuz` and the synthetic application occasionally yield when waiting for synchronization and this use of `yield` at unexpected times allows the processes to become unsynchronized. `Tpuz` vs. null works quite well and `Tpuz` vs. `Tpuz` reasonably well with this scheduler. Even though a large number of messages are buffered, the overhead of buffering is relatively small, `Tpuz` tolerates the extra latency of buffering and there are also some benefits gained from multiprogramming during some small amounts of idle time that exist due to load imbalance in `Tpuz`. The synth-N applications perform poorly, however, not because of buffering but because they require frequent synchronization.

6 Related Work

Recent architectures demonstrate emerging agreement that it is important to support both shared memory and messaging operations. Shared memory is desirable for applications with dynamic, unpredictable communication. Messaging is useful for bulk transfer of data, for combining data transfer with synchronization, and for supporting other communication models such as remote procedure call used in client/server applications. To our knowledge UDM is the first general-purpose communication model that provides efficient protected communication for fine-grained messages and directly supports a wide range of communication styles.

The UDM interface is similar to Active Messages [21]. The chief focus of Active Messages is that, since an active message is within one domain, it can carry a raw pointer to code to be executed on the receive side, thus permitting many message types to be crafted for situations. Our focus is slightly

different in that in addition to flexibility we emphasize the need for the receiving user process to have full control over atomicity with respect to messages for efficiency, since both interrupts and polling are important delivery modes. In addition, UDM provides support for running mutually-distrustful applications concurrently, allowing efficient active-message style of communication for general-purpose computing.

There are several other approaches to providing protected messaging for small messages or the equivalent functionality. These approaches can be categorized as follows:

- The machine may be divided into partitions and each partition rigidly gang-scheduled so that the network in a partition may be safely controlled by one application at a time.
- Outside of fine-grain computation, a major use for small messages is to construct synchronization primitives. Small user messages may be avoided or made rare by providing a suite of synchronization primitives in hardware, in a coprocessor, or synthesized by the kernel out of kernel messages.
- User messages may be received and demultiplexed into pre-allocated, per-application regions of physical memory. There is still the issue of how to deal with buffer overflow, but because the buffers can be large, overflow can be made to occur rarely. Messaging through memory is popular because it avoids processor modifications and decouples the delivery of the message data from the delivery of the message event.
- In a system with kernel messages, flexibility may be recovered by contriving to safety-check and then "download" user-provided message handlers into the kernel [2, 7].

Several previous multicomputers have provided direct messaging. The Mosaic [17] and J-machine [6] multicomputers are single-user machines. The J-machine provides two levels of network priorities and the ability to relaunch incoming messages from memory transparently. The CM-5 multicomputer [11] provides direct, user-level messaging and allows multiprogramming via strict gang scheduling.

Typhoon [15], *T [13] and the Meiko CS-2[16] provide coprocessors with the ability to run user code. Typhoon's coprocessor is protected by hard gang scheduling in the manner of the CM-5. The coprocessor includes features to accelerate message handling, but provides less computational performance and a restricted set of computational features (*e.g.,* no floating point support). *T proposed protection mechanisms similar to FUGU's for short messages and made the same assumptions about separating performance from correctness in scheduling. Our focus is on developing a minimal set of mechanisms that are sufficient for user-level message handling and protection, while *T proposes a richer set of features. For instance, *T demultiplexes messages into several receive queues, and the receive queues are implemented as a part of the processor register set. The CS-2 coprocessor primarily demultiplexes messages into memory but allows some user-level processing of messages, albeit with limits in speed and functionality [16].

FLASH [10] uses a kernel-level coprocessor for message handling including shared-memory protocol messages. The coprocessor implements synchronization primitives and provides user messaging through memory. Kernel messages serviced by the main processor are also implemented at greater cost [5]. Unlike FUGU, Flash does not provide general-purpose, user-level messaging except through memory. FLASH includes two networks but for a different reason than FUGU. FLASH uses its two networks for request and reply packets to avoid deadlock. Packets can be sorted into request and reply networks because the

message protocols are controlled by the kernel-level coprocessor. FUGU mostly uses one network with mixed kernel and user traffic, but occasional resorts to use of the second network to avoid deadlock.

SHRIMP [3] and Hamlyn [22] implement protected user-level message passing through a remote-write model. Communication is possible between pairs of virtual pages. Protection is provided by memory mapping but pages must be pre-negotiated and pinned.

Work on parallel processing on networks of workstations seeks to identify mechanisms to accelerate communication while retaining protection. Thekkath's RMA model [19] separates data transfer from notification in a network of workstations. The data transfer can be protected by memory mapping mechanisms. The uNET system [20] provides support for short, user-level messages in a network of workstations by demultiplexing messages into pre-negotiated, per-process receive buffers in memory. The arrival of messages is detected by polling. uNet uses the ATM VCI field in a manner similar to FUGU's GID field. The network of workstations work, with round trip times on the order of 60+ μS, is really operating in a completely different domain from Fugu's. They are in a domain where adding 6μS to the common case path to deal with unreliability in the network is comparatively small. In contrast, Fugu is operating in the domain where fetches from main memory, such as are required in models that deliver messages to memory, become significant.

7 Conclusion

The UDM system combines support for the requirements of client-server decomposition and multiple protection domains with efficient, user-level access to hardware that permits specialization. The programmer's model is that of user control of a dedicated network, while the implementation transparently provides two cases, one with actual user control over the hardware for best case performance and a second using software buffering that maintains guaranteed operation in the face of uncertainty introduced by usability features (multiprogramming and virtual memory). The system achieves high performance when co-scheduling is successful.

The novelties of the system are three: First, the programmer's model recognizes atomicity explicitly, giving user code the same level of control found in kernel device drivers. Second, the revocable interrupt disable mechanism in the hardware allows user code to control the network in the fast case but allows the operating system to manage the transition to software buffering when required. Finally, the buffering strategy uses virtual buffering and overflow control to guarantee message delivery without adding overhead to the fast path. We have presented a single, integrated design for clarity but, in fact, the independent features of the design can be applied separately:

- The message receive model with explicit atomic sections is applicable to any fine-grain messaging system. Although we have assumed an environment of a multiprocessor with a reliable internal network, the receive model is also useful on a network of workstations with an unreliable network because it allows the user to construct only as much of a software layer as is needed to provide sufficient reliability.
- The revocable interrupt disable mechanism and hardware support is applicable to any system that provides fast, protected upcalls from kernel to user code. This technique offers an alternative to downloading user code into the kernel as is popular in recent user-extensible operating systems.
- Virtual buffering and overflow control are applicable to systems with hardware-provided receive-

side buffering as well as software buffering systems as a means to provide graceful degradation as buffers fill up.

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 (ISCA'95)*, pages 2–13, June 1995.

[2] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[3] 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 (ISCA'94)*, pages 142–153, April 1994.

[4] 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 (SPAA'95)*. ACM, 1995.

[5] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[6] 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.

[7] Dawson R. Engler, M. Frans Kaashoek, and Jr. James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[8] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Fifth Internataional Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Boston, October 1992. ACM.

[9] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Supercomputing Conference (ISC) 1993*, Tokyo, Japan, July 1993. IEEE. Also as MIT/LCS TM-498, December 1992.

[10] 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 (ISCA) 1994*, Chicago, IL, April 1994. IEEE.

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

[12] 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.

[13] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 156–167. ACM, 1992.

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

[15] 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 (ISCA) 1994*, Chicago, IL, April 1994. IEEE.

[16] 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.

[17] 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.

[18] Patrick G. Sobalvarro and William E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Lecture Notes in Computer Science, number 949*, Santa Barbara, 1995. Springer Verlag. Workshop on Parallel Job Scheduling, IPPS '95.

[19] 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.

[20] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM, December 1995.

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

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