Remote Queues: Exposing Message Queues for Optimization and Atomicity*

Eric A. Brewer
UC Berkeley

Frederic T. Chong
MIT

Lok T. Liu
UC Berkeley

Shamik D. Sharma
University of Maryland

John D. Kubiatowicz
MIT

We introduce Remote Queues (RQ), *a communication model that integrates polling with selective interrupts to support a wide range of applications and communication paradigms. We show that polling is desirable for a range of applications for both performance and atomicity. Polling enables optimizations that are essential for fine-grain applications such as sparse-matrix solution. Polling also improves flow control for high-level communication patterns such as transpose.*

We use RQ to implement active messages, bulk transfers, and fine-grain applications on the MIT Alewife, Intel Paragon and Cray T3D using extremely different implementations of RQ. RQ improves performance on all of the machines, and provides atomicity guarantees that greatly simplify programming for the user. RQ also separates handler invocation from draining the network, which simplifies deadlock avoidance and multiprogramming.

We also introduce efficient atomicity mechanisms on Alewife to integrate polling with interrupts, and discuss how to exploit interrupts on Alewife and the Intel Paragon without forfeiting the atomicity and optimization advantages of RQ.

1 Introduction

Active messages, as developed by von Eicken *et al.* [vE+92], are a powerful primitive for fine-grain communication. In particular, they can integrate incoming message data into the receiver's computation by interrupting the receiver and executing a handler.

Unfortunately, to avoid the high cost of interrupts, polling for messages has emerged as the default communication model on machines such as the CM-5 [Lei+92]. For example, the Strata communication library is based entirely on active messages with polling [BK94]. In this paper, we present evidence that polling is desirable even in the presence of fast interrupts, such as on the MIT Alewife machine. Polling provides several advantages over interrupts including new opportunities for optimization, precise control over incoming traffic that enables better flow control, and atomicity by default, which greatly simplifies correct parallel programming.

We introduce the Remote Queue model (RQ) as a communication model equivalent in power (in its simplest form) to active messages with polling; it is thus simpler than full active messages. In essence, we separate the *queuing* of messages from the *invocation* of handlers. This enables new optimizations, provides a clean atomicity model, and simplifies implementation and multiprogramming.

We use RQ to implement active messages, bulk transfers, and fine-grain applications on the Intel Paragon and Cray T3D using extremely different implementations of RQ. We also show how polling on the CM-5 and Alewife maps to the RQ model. RQ enables improved performance on all of the machines.

Many machines, however, rely upon message interrupts to provide protection, support system functions, and support advanced memory models. Message interrupts support both the global memory of the Cray T3D and the coherent shared memory of Alewife. These machines also rely upon interrupts to service system messages. Thus, we integrate RQ with support for selective interrupts, which allows full implementations of active messages. We introduce efficient atomicity mechanisms on Alewife to integrate RQ with interrupts, and discuss how to exploit interrupts on Alewife and the Paragon without forfeiting the atomicity advantages of RQ.

We also show how RQ supports higher level communication such as block transfers and transpose patterns. In particular, we integrate RQ with DMA, exploit RQ for better flow control, and show how RQ allows new optimizations for strided block transfers and bulk communication patterns such as transpose.

Finally, RQ simplifies multiprogramming in the presence of communication coprocessors, which include host-interfaces for networks of workstations. In particular, RQ allows multiprogramming with unrestricted user-level handlers and without strict gang scheduling. In contrast, the CM-5 and Typhoon [RLW94] both rely upon gang scheduling to isolate the system from user failures to drain the network, while FLASH prohibits user-level handlers to prevent deadlock [Kus+94].

We evaluate RQ on four platforms, the CM-5, Intel Paragon, Cray T3D, and the MIT Alewife. Table 1 summarizes the key aspects of each one. Figure 1 illustrates their general node architecture. We use the CM-5 to motivate the RQ model and cover the rest of the platforms in Section 3. The CM-5 is a distributed-memory multiprocessor based on 33-MHz SPARC processors. The topology is a 4-ary fat-tree [Lei+92] with custom routers and network interfaces. It also provides a control network for barriers, scans/reductions and broadcasts.

Platform	Topology	Proc/MHz	Messaging Unit
CM-5	64-node Fat Tree	SPARC/33	None
Alewife	8x4 Mesh	Sparcle/20	Cache Coherent DSM + MP + DMA
Intel Paragon	4x2 Mesh	i860/50	i860 + DMA, Cache Coherent Bus
Cray T3D	8x8x4 Torus	Alpha/150	Global Addressing DMA Fetch-and-Increment

Table 1: Summary of the Studied Platforms

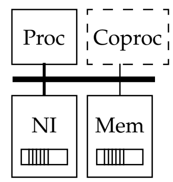

Figure 1: General node architecture. All platforms have a microprocessor, network interface (NI), and memory, generally all on a memory bus. The Paragon has an i860 coprocessor. The Alewife's NI is on the cache bus.

The rest of this section reviews active messages, defines RQ, and introduces the fine-grain sparse-matrix application that we use as a motivating benchmark. Section 2 covers our goals and how RQ attains them. In Section 3, we examine three implementations of RQ in detail: the Paragon, the T3D, and the Alewife. Section 4

*: Eric Brewer, brewer@cs.berkeley.edu, is supported in part by TITAN and Mammoth, NSF grants CDA-8722788 and CDA-9401156. Fred Chong, ftchong@lcs.mit.edu, and John Kubiatowicz are supported in part by ARPA contract N00014-94-I-09885, in part by NSF Experimental Systems grant MIP-9012773, and in part by an NSF PYI Award to Anant Agarwal. Lok Liu is supported by DOE grant DE-FG03-94ER25206 and by TITAN/Mammoth. Shamik Sharma is supported by NASA grant NAG-11560, by ONR grant SC 292-1-22913, and by ARPA grant NAG-11485. CM-5 support is from Project SCOUT, ARPA contract MDA972-92-J-1032.

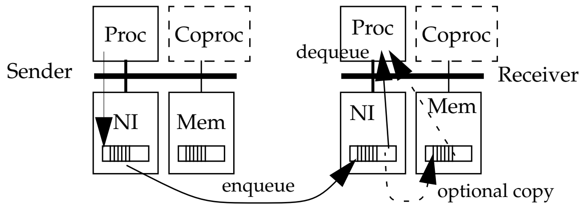

Figure 2: Implementations of the Remote Queue (RQ) model. Queues may reside in either the NI or memory. The coprocessor may also dequeue messages.

shows how RQ supports higher-level communication operations such as transpose, while Sections 5 and 6 discuss related work and our conclusions respectively.

1.1 Active Messages

Active messages form an efficient mechanism to integrate communication into computation by associating a small amount of computation at the receiver via a handler. Each active message contains a handler address in addition to the data to be transferred. Upon arrival of the message, the ongoing computation is interrupted and the handler is invoked. The role of the handler is to quickly extract the message out of the network by consuming the data with a small amount of computation and then optionally sending a reply. Active messages integrate data and control transfer in one primitive on top of the network interface. To avoid deadlock and livelock, active messages must use restricted handlers, since handler invocation blocks incoming traffic.

1.2 The Remote Queue Model

We introduce *Remote Queues* (RQ), a simple model that provides a general abstraction for low-level communication. RQ consists of three basic elements. First, one or more *named queues* at a receiving node. Second, an *enqueue* operation:

enqueue(n, q, arg0, ... argn, sbuf, nbytes) → *true* | *false*

that causes *arg0* through *argn*, followed by *nbytes* bytes from the buffer, *sbuf*, to be queued in the queue named *q* on node *n*. *enqueue* immediately returns *true* if the action is committed and *false* otherwise. The action is *committed* if the system guarantees that it will complete. Third, a *read_queue* operation:

dequeue(q, arg0, ..., argn, rbuf, nbytes) → *true* | *false*

that causes a node to remove *arg0* through *argn,* followed by *nbytes,* from its queue named *q*; and places them in *arg0...argn* and *rbuf* for processing by the processor (or coprocessor). *dequeue* returns *true* if the first queue item is equal to or larger than the amount of data requested, it returns *false* otherwise. It is the user's (or compiler's) responsibility to provide *dequeue* arguments that make sense for the head of the queue, although the user can look at part of the message to handle this dynamically. *dequeue* is a generalization of polling to named queues rather than solely the NI hardware queue. This allows variable-length, typed argument lists with integrated DMA.

Figure 4 illustrates possible implementations of the three basic elements of RQ. Additionally, it is often useful to separate the *dequeue* operation into:

read_queue(q, arg0, ..., argn, rbuf, nbytes) → *true* | *false*

delete_queue(q) → *true* | *false*

where *read_queue* has the same semantics as *dequeue* but does not modify *q*. *delete_queue* removes the first queue item from *q*, either returning *true* if successful or *false* there was no item to remove.

As shown in Figure 4, implementing polling-based active messages on top of RQ is straightforward. However, active messages

```
rq_send_am(proc, handler, arg1...argn)
    enqueue(proc, am_q, handler, arg1...argn)

rq_poll_am()
    if (dequeue(am_q, handler, arg1...argn))
        (*handler)(arg1...argn);
```

Figure 3: Implementing polling-based active messages on RQ. The number of arguments (*arg1...argn*) is generally fixed in active message implementations.

with interrupts are more powerful than the version of RQ presented so far. First, interrupts drain the network even if the user forgets to dequeue (see Section 2.2). Second, for infrequent arrivals the overhead of failed dequeue calls leads to worse performance than simply using interrupts. Third, interrupts are often essential to the correct operation of a system; for example, both Alewife and networks of workstations require interrupts for system traffic.

To address these issues, we use *selective interrupts* in conjunction with RQ. Selective interrupts mean that an interrupt is generated only for specific messages (such as system messages) or under specific circumstances, such as network overflow. Other RQ messages are queued, either in the NI or memory, for later dequeuing rather than handled immediately during an interrupt.

In general, RQ is a single abstraction for a continuum of queueing strategies from all hardware (queues in the NI) to all software (queues in memory). A mixture of hardware and software strategies is possible and is used for Alewife. Using hardware queues on an NI minimizes data copying, but limits the size and number of queues. Using queues in memory increases the size and number of queues, but requires the processor, coprocessor, or DMA engine to copy messages from the NI to memory.

1.3 An Inherently Fine-Grain Application

We drive many of our experiments with an important class of fine-grain applications, computations that involve iterative sparse-matrix solution. Although these applications are extreme in their fine granularity, they are both an important class in themselves and serve to emphasize properties also present in many data-parallel and High-Performance Fortran (HPF) programs. Additionally, although our experiments often use hand-optimized, low-level code, our communications techniques are intended for compilers such as those for HPF or data-parallel languages.

Iterative solutions are widely applicable to sparse matrices from both scientific and business problems. The key issue is that iterative solutions all use some form of incomplete factorization that reflects the underlying structure of the sparse problem. Unfortunately, many real-world problems have a sparse and irregular structure. There are no known techniques for organizing such irregular problems (with directed edges) into coarse-grain computations.

The techniques of Chong *et al.* [CSBS95] have produced unprecedented performance on sparse triangular solution through substitution, the dominant computation in sparse iterative solution. To evaluate communications mechanisms on real applications, we use their methodology on several power-grid problems from the Harwell-Boeing benchmark set [HB92]. In this paper, our examples are *bcspwr05*, a square matrix with 274 rows and 759 non-zeros; and *bcspwr06*, a square matrix with 1454 rows and 1923 non-zeros.

Most of the computation in these applications occurs in message handlers and communication overhead is 30–40% of total execution time. Every 10 cycles we cut from communication overhead results in roughly a 5% improvement in application performance.

2 Goals for RQ

In this section we define the goals of the RQ model, describe the key obstacles to achieving these goals, examine how current techniques approach these issues, and discuss how RQ achieves each goal. We define goals from four categories:

- **Benefits of polling**: efficient fine-grain communication; atomicity between user code and incoming messages; and support for global communication patterns.
- **Benefits of interrupts**: forward progress; and support for asynchronous communication.
- **Exploiting coprocessors**: support for multiprogramming; specialized handlers; and protection.
- **Additional goals**: integration with DMA and block transfers; and deadlock avoidance.

These goals are not obviously complementary. In particular, the forward progress advantages of interrupts appear to conflict with our desire for the clean atomicity model of polling. However, we will show how selective interrupts achieve both the advantages of polling and interrupts for RQ.

2.1 Achieving the Benefits of Polling

Since dequeue is just a controlled form of polling, RQ inherits the following benefits of polling: efficient fine-grain communication, atomicity by default, and support for global communication.

Efficient Fine-Grain Communication: Traditionally, receiving a message via an interrupt costs substantially more than receiving through polling. Even on the relatively modern CM-5, receive overhead for polling is tiny compared to that for interrupts [BK94]. For example, with CMMD 3.1, the overhead of receiving one message per poll is 3.3 µs (100 cycles), while the overhead of receiving one message per interrupt is 19 µs [LC94]. By streamlining the code and minimizing the saved state, we were able to reduce the interrupt overhead to 9.6 µs, which is still 3 times worse than polling.

There is also significant cost to enabling and disabling interrupts. Since the network interface (NI) interrupt-enable bit can only be written in supervisor mode, enabling and disabling message interrupts require system calls. Enabling message interrupts takes 4.9 µs and disabling takes 3.8 µs.

However, it is possible to reduce interrupt overhead to a level comparable to polling. The MIT Alewife machine, through tighter processor-NI coupling and a reserved register set, can receive with an interrupt in 55 cycles. This cost is comparable to polling, which takes 45 cycles (see Section 3.2 for details). For most machines, polling remains significantly faster than interrupts.

A subtle difference between polling and interrupt-based communication is the required handler invocation in the interrupt model. Although the general polling loop on Alewife is comparable in overhead to an interrupt, the user can further optimize the polling code when there is some knowledge about the incoming messages, which is common for global communication patterns such as transpose. When a small number of message handlers are used in an application, polling allows these handlers to be inlined in the polling loop. On Alewife, this optimization can save about 10 cycles and results in up to a 5% improvement in the performance of our sparse triangular solutions (see Section 3.2.5), which only use one kind of message handler. Strata's radix sort [BK94] on the CM-5 can redistribute the keys among the nodes 13% faster by dequeuing the packets in a tight loop and moving the keys to their correct location, than by invoking active message handlers for each message.

In fact, the polling loop itself can be optimized in many cases, cutting overhead from 45 to as little as 14 cycles for a hand-coded assembly loop on Alewife. Within the loop, other optimizations are available, such as keeping synchronization variables in registers. Polling exposes both communications control and code to the user, allowing optimizations unavailable with interrupts.

Atomicity: In sparse triangular solution, for example, each variable of the solution vector must accumulate several floating-point values atomically, in addition to the atomic decrement of an associated synchronization variable. These operations are implemented efficiently in message handlers on both the CM-5 and Alewife. In the polling implementations on these machines, all code, including message handlers, runs atomically except at explicit polling points. In the interrupt-driven implementation, interrupts must be explicitly enabled and disabled around critical sections.

In many semi-synchronous applications, explicit polling is easier to insert than explicit atomicity. Polling naturally belongs at barrier-synchronization points common in data-parallel and High-Performance Fortran codes. Such points also occur in iterative computations such as our sparse triangular solver.

Anecdotal evidence suggests that inserting explicit polling is often far easier than inserting explicit atomicity. Polling errors lead to deadlock or poor performance, which can be resolved using traditional debugging tools. With interrupts, the common error is missing protection for critical sections, often because the user did not realize the section had to be atomic. Unfortunately, these errors are extremely difficult to track down: the bug is nondeterministic and the symptoms may appear far after the problem and in apparently unrelated code. A common example is corrupted data structures: it is clear the structures are corrupted, but totally unclear as to the cause. Worse, it is often impossible to reproduce the problem reliably. Atomicity problems can be further obscured by handlers, macros, or compiler optimizations. Message handlers execute atomically on most systems, but explicit atomicity is required when a handler is called as a local procedure to avoid a self-message send. A compiler will often remove a reference to a synchronization variable unless the variable is declared *volatile*.

The atomicity advantages of polling were recognized by Spertus *et al.* [SPE+93]; they simulated polling on the J-Machine by disabling interrupts and only temporarily enabling interrupts at "polling" points. We call this use of interrupts *pseudo-polling*. Unfortunately, the use of interrupts by the OS normally prevents users from disabling interrupts; we develop techniques for safely providing pseudo-polling in Section 3.2.

To summarize, the fundamental advantage of polling for atomicity is that code *defaults* to being atomic. With interrupts, code is not atomic unless explicitly protected. Thus, when users forget or are unaware of critical sections, polling leads to working code, while interrupts lead to nondeterministic bugs. The communication layer should aim for the robust behavior of polling.

Support for Global Communication Patterns: Polling allows the system to schedule communication precisely, by allowing the receiver to temporarily put off accepting a message. With interrupts, the receiver cannot ensure progress on sending other than by turning off interrupts temporarily. This increases the overhead and leads to burstiness that reduces the overall bandwidth of the system. Section 4 covers this effect in detail and shows how using RQ to limit reception allows faster block transfers.

2.2 Achieving the Benefits of Interrupts

Forward progress: Unfortunately, polling is often difficult to integrate into a full system. In particular, it is difficult to maintain protection and keep the network clear in a polling model. Although polling is easy for barrier-synchronized applications, it is extremely difficult to know when to poll in asynchronous communication such as when maintaining cache coherence in a shared-memory system. In general, interrupts ensure that critical systems communication occurs in a timely fashion. A low-level communication layer should provide some reasonable guarantees on response time and forward progress for system communication. Pure polling-based models require that users ensure these properties.

Alternatively, we can implement RQ such that messages in the NI may be transparently moved to queues in user memory in order to drain the network and thus ensure progress for system traffic. This has two consequences: first messages may be handled out of order, since system messages execute immediately while user messages are queued. Second, user messages are only atomic with respect to each other and the primary user computation, system messages execute independently of polling points. This model thus combines the default atomicity and performance of polling with the need for interrupts for system traffic. The only possible downside is the occasional need to copy packets out of the network. Note that we only need to copy packets out of the network if a system packet caused an interrupt or if system packets have been delayed in the NI. On Alewife, for example, we only move the queue to memory if the NI queue has not made progress within a specified timeout interval (see Section 3.2.3).

Asynchronous Communication: Unstructured parallel programs, such as those that use concurrent objects, typically cannot tell when messages will arrive (unlike SPMD programs). For these applications, polling can lead to substantial overhead if the frequency of arrivals is low enough that the vast majority of polls fail to find a message. With interrupts, overhead only occurs when there are arrivals. Thus, for infrequent arrivals, we may want to avoid the overhead of polling, even if we must forfeit some of the atomicity advantages of polling.

As mentioned in Section 1.2, RQ can avoid this problem by selectively allowing interrupts on infrequent messages. This forfeits default atomicity for at least certain types of messages and handlers, but may be worthwhile. It differs from the system-traffic case above, in that the latter only forfeits atomicity between user and system handlers.

2.3 Exploiting Coprocessors

Support for Multiprogramming: To understand the issues for multiprogramming on coprocessors, we must cover the difficulties that user-level active messages introduce. First, it is relatively easy to deadlock the network with user-level handlers. In the CM-5 and Typhoon [RLW94], this is allowed but is isolated from other users by strict gang scheduling that includes draining the network during context switches. Thus, these systems provide deadlock avoidance at the expense of multiprogramming. FLASH [Kus+94] prohibits user-level handlers to avoid deadlock, which allows true multiprogramming but may reduce the usefulness of the coprocessor.

The key observation is that executing handlers and draining the network are currently coupled only by convention (as in active messages). User-level handlers in RQ can be split across the two processors: the coprocessor *drains* the network into queues, while the primary processor *executes* handlers via the dequeue operation. As long as we drain the network, then there are no restrictions on handlers. Furthermore, if we separate the two aspects, it becomes clear that we can drain messages into named queues in the address space of any user, not just the currently executing user, since we need not invoke the handler immediately.

Specialized Handlers: Although RQ gives us powerful user-level active messages, there is a performance hit for running the handlers on the main processor rather than the coprocessor. There is also added latency, since replies are delayed until the user code executes. Fortunately, there is a simple solution: place the most common handlers on the coprocessor via specialized handlers, as proposed in FLASH [Kus+94]. The only requirement on specialized handlers is that they avoid deadlock. These handlers may include read and write operations, barrier and scan/reduction operations, and atomic memory operations. In fact, *enqueue* can be viewed as a specialized handler that enables user-level active messages.

On the Paragon, specialized handlers, including enqueue, have been added to the kernel on the coprocessor. Besides improving performance, this arrangement was *required* since the coprocessor prohibits user-level code. Despite this limitation, we achieve both excellent performance and fully general user-level handlers.

Queues and Protection: With a coprocessor, the queues can either go into user memory or can correspond to actual queues in the coprocessor or network interface. There are advantages to each.

With queues in user memory, there is essentially no limit on the number of distinct queues, each application manages its queue memory independently, and there is almost always space for incoming messages. This solution is used in the Paragon implementation.

With network queues, a small number of queues must be shared among all applications. This sharing must occur in such a way that the queues appear independent; that is, the traffic from one application should not receive a disproportionate share of network resources or have the ability to inhibit or deadlock another application. With network queues, we must either use gang scheduling to isolate the applications, or copy the packets into user queues with trusted code. Examples of network queues include Alewife, Myrinet [Sei94] and the SP/2 communication adaptor [Stu+94].

The advantage of network queues is performance; in particular, the producer-consumer relationship of the coprocessor (or NI) and the main processor fits poorly with the invalidation protocols used by the cache-coherent buses. With the queues in user memory, we achieve poor cache performance in both directions and we force the data to move twice: first into memory and then into the main processor. An update protocol would greatly reduce this problem. Note that the extra copying does not apply to DMA blocks, which are moved directly to their destination instead of to the queue.

When we have many queues in user memory, we can also use the queues to provide protection by associating process IDs with each queue. Thus, the queue becomes the vehicle for naming and for protected communication. The *T design used this approach to detect invalid enqueues, which then caused a kernel trap [PBGB93]. This form of naming and protection also makes sense for networks of workstations, but we are still investigating these issues.

2.4 Additional Goals

Integration with DMA: Many current multiprocessors, including Alewife, the Paragon and the T3D, provide direct-memory access (DMA) for block transfers. There are important concerns when integrating DMA transfers with the basic communication layer. First, on the sender side, we would like to know when the access to the outgoing buffer is complete so that we can reuse the buffer. Second, we want to inform the receiver when the incoming transfer is complete, so that appropriate action can be taken. This is similar to invoking a handler at the receiver, except it must imply that the transfer is done. We also want to avoid excess copying: the block should go directly from user memory to user memory.

RQ can notify the sender for buffer reuse by simply posting a message into the incoming queue of the sender. RQ can notify the receiver of transfer completion by posting a message into the receiver's queue in one of two ways. We can either post the message after the transfer is complete, or we can post the message after it has started and provide a second function (to be called by the handler) that waits for completion. Although the former is simpler, the latter allows the processor to overlap work with the incoming transfer; we use this technique on Alewife and it seems preferable. In both cases the handler can easily tell when the DMA is done.

Deadlock Avoidance: Deadlock avoidance is one of the most difficult issues of multiprocessor networks. Typically, the network avoids deadlock if and only if the user regularly drains the network. There are several ways that the user can fail to meet this contract and cause deadlock. For example, if the user is polling and enters an infinite loop, then that node will fail to drain the network and deadlock may ensue. This is particularly unacceptable if it interferes with other users or with system communication.

A more complex case is the *fetch-deadlock* problem [Lei+92], in which two nodes block trying to inject traffic. If at least one of them drains the network, deadlock is avoided. The common solution (used in the CM-5, Typhoon, and FLASH) is to divide the network, physically or logically, into *request* and *reply* halves and ensure that the reply half can always drain the network.

Thus, the low-level communication layer must ensure that the network drains correctly even if a user forgets to poll. This will ensure forward progress for system communication and isolation among users without strict gang scheduling.

RQ does not need request-reply protocols. The key to deadlock avoidance in RQ is the observation that we can drain the network without invoking any handlers. This can be done with a coprocessor or after an interrupt due to the arrival of a system message.

2.5 Summary

The Remote Queue model exposes the underlying network queue structure to exploit the atomicity and optimization benefits of polling. By augmenting RQ with selective interrupts we allow integrated use of polling and interrupts, which is required on systems that depend on interrupts for correctness. Finally, RQ provides additional benefits for coprocessors, including unrestricted user-level handlers without gang scheduling and a convenient way to name network resources and provide protection.

3 RQ Implementations

In this section, we describe our experiences using the RQ abstraction as an implementation tool on the Paragon, T3D, and Alewife machines. Each section discusses (in order):

- system characteristics,
- deadlock avoidance,
- flow control and queue management,
- integration of RQ and block transfers, and
- overall performance.

3.1 Intel Paragon

Each node of the Intel Paragon is a shared-memory multiprocessor with up to 3 Intel i860XP microprocessors and a DMA engine connected by a cache-coherent bus. The nodes are connected by a 175 MB/s 2D mesh to form a distributed-memory multiprocessor. Each node runs the Paragon OSF/1 AD operating system to provide a single system image. Since the network interface does not distinguish between system and user messages, one of the i860s is designated as the *message processor* and runs the message handling part of the kernel to provide protected communication. As a result, the user cannot access the NI FIFOs directly. The current OS does not support user-level handlers on the coprocessor.

3.1.1 Implementing Active Messages on top of RQ

As demonstrated by the Paragon Active Message (PAM) implementation, the RQ model is a simple and flexible abstraction on which active messages can be built. Rather than writing a new operating system as in SUNMOS [MMRW93], the PAM implementation is built within the Paragon OSF/1 framework so that it can coexist with the existing Intel libraries such as NX [Pie94] and kernel IPC. PAM consists of two parts. The kernel half is a message module providing the basic enqueuing, dequeuing, and DMA operations on the message processor. The user half is a library that implements the PAM functions using the RQ model on the compute processor.

The interface between the compute processor and the message processor is abstracted as a send queue and a receive queue in shared memory. As a result, the interaction between the compute processor and the message processor is reduced to a producer-consumer problem. The producer has a write pointer that points to the next entry to which it can write. Similarly, the consumer has a read pointer that points to the next entry from which it can read. Using separate pointers avoids the required critical section for updating a single pointer. Each entry has a presence flag to indicate whether the entry is valid for reading. To write an entry, the producer checks that the presence flag is not set, writes the data, and sets the flag. To read an entry, the consumer checks that the presence flag is set, reads the data, and resets the flag.

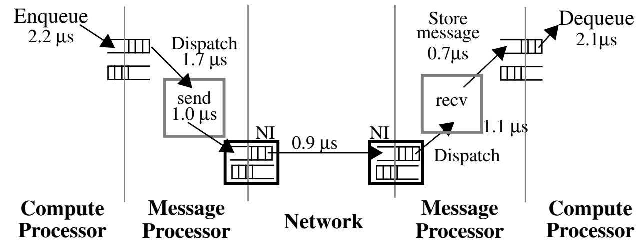

Figure 4: RQ Costs on the Paragon

As shown in Figure 4, to send an active message the user program enqueues a 7-word message into the send queue consisting of the handler address, 5 words of data and a message tag. The message processor, which is spinning in a dispatch loop, picks up the send request and sends out the message when the network is ready. When the message arrives at the receiver, the message processor reads in the message from the network interface FIFO and stores it into the appropriate receive queue. To poll an active message, the user program can simply dequeue the message from the receive queue. Since the ABI of the i860 does not allow more than 2 words to be returned from a function, the dequeue operation is integrated with the active-message dispatch loop. The active-message polling function deposits the data directly from the receive queue to the argument registers before invoking the handler function.

3.1.2 Deadlock Avoidance

To avoid deadlock, we must ensure that the message processor is always receiving even when the receive queue is full. In PAM, the number of entries in the receive queue is specified by the user program during initialization. To ensure that the message processor code is deadlock free, the message processor promises not to drop any messages as long as the user avoids overflowing the receive queue. The message processor currently relies on the user programs running to implement their own flow control to prevent the receive queue from overflowing.[1] Since the message processor is always receiving, the send queue will eventually be emptied even if the compute processor does not poll the receive queue.

3.1.3 Flow Control and Queue Overflow

The underlying RQ implementation avoids restrictions on the handlers as long as the user ensures that the receive queue does not overflow. However, PAM implements simple window flow control with a request/reply protocol in the user-level library so that we can focus on implementing the kernel code and ensure that the PAM library is compatible with existing active-message implementations. We can avoid the protocol if we know something about the traffic; for example, transpose has no need for window flow control.

In this scheme, the receive queue is divided equally among all the nodes. Sending a request uses up a credit to send. Receiving the corresponding reply replenishes a credit. If the sender uses up all of its credits, it will start polling until it can send. If the request handler does not generate a reply, the PAM layer will automatically generate a null reply to the sender. Note that the request/reply protocol is used here to avoid overflowing the receive queue rather than to avoid deadlock. The advantage of this approach is that it avoids the overhead of a round-trip time for fetch-and-add or acknowledgment. The disadvantage is that the queue size scales with the number of processors and the depth of the network.

1: Alternatively, the message processor can interrupt the compute processor to drain the receive queue as in Alewife (see Section 3.2.1).

3.1.4 Block Transfers

The size of a queue entry is typically a cache line, so RQ by itself is inefficient for large transfers. In PAM, we combine DMA transfer with RQ so that computation can be overlapped with communication. The notification of the completion of DMA transfer on both the sender and the receiver is done through the RQ mechanisms. The handler is included in the header of the message and is sent along with the bulk data.

Two primitives are supported: `am_store_async` and `am_get`. `am_store_async` performs an non-blocking write to a memory region of a remote node. The compute processor makes a request to the message processor by storing a descriptor into the send queue. The message processor then fragments the data into packets, sends out the header, performs the address translation, and initiates the DMA engine to transfer the data directly from user memory to the NI. Upon completion of the DMA transfer on the sender, the message processor enqueues a handler onto the *sender's* receive queue. On the receiver, the message processor transfers the data from the network directly into user memory using DMA and enqueues a user handler onto the *receiver's* receive queue.

Similarly, `am_get` performs a non-blocking read from a memory region of a remote node. In this case, the local node makes a request to the remote node, which replies using DMA. At the receiver, if the message processor is not sending, the message processor can reply directly to the sender without intervention of the compute processor. Otherwise, the message processor stores the request into the receive queue of the compute processor for later processing. After the requester receives the data using DMA, a handler is enqueued into its receive queue.

3.1.5 Integration of Interrupts

On the Paragon, the message processor can trigger an interrupt on the compute processor. In addition to the possibility of generating an interrupt when the receive queue is full, if we expect infrequent message arrivals, we can tag the message so that the message processor triggers an interrupt. We expect to add selective interrupts to RQ eventually, but so far they have not been needed.

3.1.6 Specialized Handlers

With a message processor, specialized user handlers can be executed on the message processor instead. For example, Split-C makes use of counters to synchronize split-phase writes (`put` or `store`) and reads (`get`). By updating the counter directly on the coprocessor, we can save both the 2.8 µs required to hand a message from the coprocessor and the 2.1 µs required to dequeue. For `get`, we also reduce the round-trip time. Other examples of specialized handlers include scans, and strided block transfers.

3.1.7 Performance

The preliminary version of PAM is now running on an 8-node Paragon XP/E with two i860 processors per node and a 175 MB/s network. Split-C [CDG+93] is also running on top of PAM. The one-way total latency of enqueuing and dequeuing a message is 9.6 µs. Building active messages on top of RQ increases the one-way latency slightly to 10.1 µs. Figure 4 shows the breakdown of the one-way latency of RQ. Enqueuing and dequeuing messages in a pipeline takes about 2 µs per message, with about 1 µs due to cache invalidations. The hand-off between the compute processor and the message processor increases the round-trip latency.

For block transfers, `am_store_async` has an overhead of 2.5 µs on the compute node. The one-way latency to transfer 64 bytes (the smallest DMA) is about 26 µs, while the peak bandwidth is about 142 MB/s. Table 2 shows the breakdown of the overheads for a 64-byte data transfer. Note that even though the expected timer overhead has been subtracted, the overheads in Table 2 still add up to more than the 26 µs latency because reading the off-chip timer interferes with the operation of the code. Overall, sending overhead is the bottleneck. The peak bandwidth is limited by the overhead to packetize the data into 8KB packets, to send out the header, and to perform the virtual to physical address translation and alignment check.

`am_store_async()`	Send	Receive
Post request to coprocessor	2.5	—
Dispatch	1.2	1.0
Look up route + check NI FIFO	0.5	
Receive rest of header	—	1.4
Validate address	1.4	1.4
Misc. overhead (e.g. function calls)	2.4	1.8
Send header	2.6	—
Alignment and page boundary checks	2.2	2.0
DMA transfer (64 bytes)	2.0	1.9
Enqueue handler	2.0	2.0
Poll	2.0 µs	2.0 µs
Total	**18.8 µs**	**13.5 µs**

Table 2: Breakdown of the send and receive overheads for `am_store_async` on top of RQ on the Paragon.

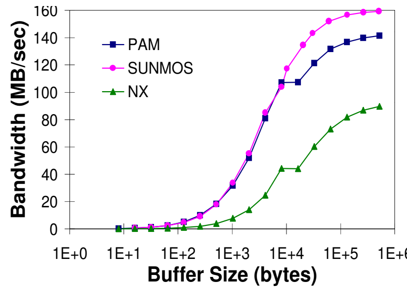

Figure 5: These graphs plot the overall bandwidth on the Paragon for PAM, Intel's NX library, and SUNMOS

Figure 5 compares the bulk-transfer performance of PAM, NX, and SUNMOS. NX is Intel's message-passing library. SUNMOS is a lean customized OS optimized for message passing. To obtain the bandwidth, we measure the time from which the send routine is called until all the data is received on the destination node. For PAM, we use the `am_store` call. For both NX and SUNMOS, an asynchronous receive is posted before the matching send to obtain the best performance. We can see that PAM achieves higher bandwidth and lower latency than NX. NX has lower bandwidth because it fragments messages into 2KB packets in the current release. Comparing to the SUNMOS message passing library, PAM has similar performance for message up to 8KB. SUNMOS can achieve higher bandwidth than PAM for the following reasons:

- SUNMOS does not support demand paging, allowing it to send a large physically contiguous block of data using 16KB DMA transfer. Larger DMA transfers allow SUNMOS to better amortize the overhead.
- SUNMOS sends data in one large packet, thereby avoiding the overhead to send out the header for each 16KB fragments. However, this can result in poor link utilization in all-to-all communication patterns in a loaded network.
- `am_store_async` is more powerful than the SUNMOS message-passing routines, since arbitrary handlers can be invoked when DMA completes at the sender and receiver.

3.1.8 Summary

Implementing only enqueue and dequeue on the message processor simplifies the implementation and makes it easier to avoid deadlock. This in turn removes restrictions on the active-message handlers. By combining DMA transfer with remote queues, we can provide high-performance asynchronous block transfers with event notification via handlers. For short messages, the latency is increased by the presence of the message processor due to copying

```
stio r2, $ipiout0  ; Store header (w/ proc)
stio r3, $ipiout1  ; Store arg0
stio r4, $ipiout2  ; Store address (sbuf)
stio r5, $ipiout3  ; Store length (nbytes)
ipilaunch 2, 1     ; Launch message.
```

Figure 6: *enqueue(proc,NIq,arg0,sbuf,nbyte),* where *NIq* is the hardware NI queue, implemented in machine code. In addition to the required header, this message includes one explicit data word, and one block of data from memory.

and cache invalidations. For block transfers, the peak bandwidth is limited by the overhead of packetization.

3.2 MIT Alewife

The MIT Alewife machine is an experimental multiprocessor that provides both distributed cache-coherent shared-memory and user-level message passing with DMA. The architecture directly supports up to 512 processors. Each Alewife node consists of a 33-MHz[2] Sparcle processor [AGA+93], 64KB of direct-mapped cache, a 4MB portion of shared memory, 2MB of private (unshared) memory, a floating-point coprocessor, and a mesh-routing chip from Caltech. The nodes communicate via messages through a direct network with a mesh topology using wormhole routing. A single-chip *Communications and Memory Management Unit* (CMMU) implements the cache coherence protocol by synthesizing messages to other nodes, and also implements the user-level message-passing interface and network queues. The hardware queue of a network interface (NI) is essentially a remote queue. This RQ view of the NI, coupled with fast atomicity mechanisms, allows us to integrate user-level polling into Alewife, which was originally designed solely for interrupt-based communication.

The Alewife message interface is designed for direct, user-level access to the network. Such access is achieved even in the face of a single level of network priority and guaranteed system-level traffic. Messages can include data directly from registers or from memory via DMA. The Alewife network coprocessor (CMMU) exports the NI directly to the computation processor via memory-mapped hardware queues and arrival interrupts. Since interrupts in Alewife can be as inexpensive as 20 or 30 cycles, interrupts are used by default for delivering messages and for providing notification on completion of DMA operations. Incorporation of polling and atomicity into this model is examined in Section 3.2.3. Note that Alewife has a hardware model that includes a much tighter coupling between network and processor than the other platforms.

An atomic launching mechanism in Alewife permits user code to compose outgoing messages directly in the hardware network queues without disabling interrupts or preventing system code from using the network interface [KA93]. Figure 6 illustrates code for sending a message complete with DMA. The store instructions illustrated here (**stio**) transfer data directly into the network message descriptor array in the CMMU.

Message reception is similarly inexpensive. Under normal circumstances, an arriving message invokes a message-arrival interrupt. This interrupt performs a dispatch on an opcode field in the message header. A special register set is reserved by the operating system for message handlers; consequently, there is no need to save or restore state to process a message. With this architecture, a system-level active message handler can begin executing in as little as 35 cycles from message arrival. Once a handler has begun execution, it can read the incoming message directly from the message input queue, or invoke DMA to store portions of the message back to memory. A user-level message handler is somewhat more expensive in Alewife, typically 50 cycles from message arrival to handler entry. Additional expense is incurred on exit, for a total of approximately 95 cycles for a null message handler. Almost 40 cycles of this time, associated with various user-level protection issues, will be eliminated in the planned respin of the CMMU (described in Section 3.2.3).

The implementation of RQ on Alewife illustrates an important point in the spectrum between hardware and software queueing. In the common case, the user has direct access to the hardware queues, thus avoiding the extra copy and cache invalidations of retrieving messages that have been copied to memory. In other situations, such as queue overflow or debugging, messages are copied to memory and later presented to the user. This allows fast user-level network access without forfeiting system-level forward progress.

3.2.1 Deadlock Avoidance

Much like the Paragon, the Alewife network hardware provides a single level of message priority with a guarantee of deadlock freedom only when network receive ports sink packets. However, instead of relying solely upon the user to prevent NI queues from overflowing, Alewife provides an additional recovery mechanism, described in the next section.

3.2.2 Flow Control and Queue Overflow

To avoid deadlock, Alewife relies upon a special "Network Overflow" interrupt that is invoked when the *output* queues have been blocked for a sufficiently long period of time. Flow control, such as the window protocol used by PAM, may also be used, but is left up to the user and is not necessary for correctness.

Once invoked, the network overflow handler pulls packets *in* from the network, storing them in local memory, until the network *output* queue begins to unclog. As soon as the network is free, the handler relaunches packets to the local node, thereby "recovering" from network deadlock. This simple overflow detection and recovery mechanism guarantees that, given sufficient local memory, network traffic can always make progress.

The same mechanism ensures that the queue can't overflow: senders are stalled until there is room. There will be room eventually since the overflow interrupt essentially moves the queue into local user memory.

3.2.3 Integration of Polling and Interrupts

Unlike the Paragon and the T3D, Alewife is designed primarily around an interrupt-driven model of execution. Unless messages are serviced at a regular rate, the machine will not execute correctly. Further, as a rule, user code should not be allowed to disable interrupts on a system-critical facility such as the network. However, both polling and atomicity are important attributes of an NI.

The following text introduces atomicity and polling mechanisms we developed for Alewife; describes *user polling*, an implementation of RQ that integrates polling and interrupts; and reports results from experiments that simulate user polling with existing Alewife atomicity and polling mechanisms.

Atomicity: To support atomicity, the Alewife runtime system makes use of two bits in a global flags register. One of these bits, *user_atomicity_request*, indicates that the user is requesting atomicity. The user sets this bit to signal the beginning of a user-level critical section. During this critical section, the runtime system will not begin execution of any message handlers. To exit a critical section, the user clears the `user_atomicity_request` bit and checks another bit, the *user_atomicity_cleanup* bit, which indicates that some end-of-critical-section cleanup is required. If set, the user executes a special cleanup software trap. Careful allocation of these two bits allows all of the exit actions (clear of the *user_atomicity_cleanup*, check of the *user_atomicity_cleanup*, and trap if set) to consume only two cycles.

On entry, a protected portion of the user-level message handler checks the *user_atomicity_request* bit, which adds a two-cycle check and branch sequence to the main path of a message handler.

2: The first version of the Alewife only runs at 20 MHz due to a bug in the CMMU. The upcoming respin will fix this problem.

Atomicity is considered *invoked* if the bit is set. The handler can do one of two things at this point to satisfy the request for atomicity. By default, it sets a timer and disables network interrupts. Alternatively, it can drain the user-level message from the network; this later option is taken in the case of network overflow. In either case, the handler sets the *user_atomicity_cleanup* bit, then returns to the interrupted user code.

Finally, when the user attempts to execute a critical-section exit, the cleanup system call is invoked, which restores the system timer and reenables interrupts. As described, the total cost of an invoked atomicity operation is about 30 cycles, while the cost of an uninvoked atomicity operation is 3 cycles. Variants of this atomicity mechanism can be applied to any hardware architecture that requires "add-on" atomicity at the user level.

Polling: It should be noted that both atomicity and user-level message protection can be handled by a single mechanism, planned for incorporation in a refabrication of the Alewife CMMU. This mechanism consists of a single bit, called the *atomicity bit,* in the Alewife interrupt controller that can be modified either by the user or the runtime system. The user can set or clear this bit, unless it is set by the runtime system, in which case the bit is "sticky" (not clearable by the user). Further, the atomicity bit is cleared when a user-level message is freed from the network queue.

When set, the atomicity bit indicates that the network is configured for user-level network control. Setting of the bit disables network interrupts only for user-level messages: system-level messages continue to generate interrupts. For protection against bad user-level code, the setting of this bit causes a timer to begin counting toward zero.[3] Should the timer ever reach zero, a special "atomicity violation" trap is invoked, and the user is faulted. If the atomicity bit is cleared before this time, the timer is reset to its original value. Thus, this atomicity bit is a type of revokable interrupt disable. Since this mechanism disables messages for shared-memory protocols, access to shared memory while the atomicity bit is enabled causes a trap to avoid deadlock; thus, this really only provides atomicity for local memory. The atomicity bit will reduce the cost of user-level message protection by almost 30 cycles and will reduce the cost of user atomicity (invoked or not) to 2 cycles.

A closely related mechanism for user-level polling will involve another interrupt-controller bit, called the *user-polling bit*, also modifiable by the user. Similar to the atomicity bit, the polling bit will disable user-level network interrupts and prevent accesses to shared memory. However, freeing a message will not clear the polling bit. Further, the countdown timer will only be enabled when a user-level message is actually *waiting* in the network interface. Thus, the user program will be faulted only if it fails to free messages at a regular rate. A simple emulation of this mode is described in the next section.

Although the current design essentially forces the user to choose between polling-based message passing and shared memory, we can generalize the mechanism by draining user messages into local memory when the polling bit is set *and* the timeout expires. With this approach we can use shared memory since the network drains regularly, but user-level messages (other than for shared memory) are queued instead of handled, which preserves atomicity. The queued messages are handled at the next dequeue. Although this allows shared-memory access, only local memory maintains atomicity, since the protocol packets are not delayed. This is similar to the Paragon implementation, which does not have shared memory, except that since the queue is normally in the CMMU, it only moves to memory as needed to drain the network.

In general, the complexity of these mechanisms derives from the integration of message passing and shared memory and from their "retrofit" nature.

3: Note that this timer will stop counting on entrance to supervisor mode.

```
ldio $ipiout0, r2    ; Load header
ldio $ipiout1, r3    ; Load arg0
ldio $ipiout2, r4    ; Load address (sbuf)
ldio $ipiout3, r5    ; Load length (nbytes)
ipicst 0, -1         ; delete message.
```

Figure 7: *dequeue(NIq, arg0, buf, nbyte)* on Alewife

```
rq_poll_inlined_augmented()
    if (read_queue(q, handler)
        if (handler==usr_add)
            dequeue(q, handler, offset, data);
            /* inlined handler */
            array[offset].outstanding++;
            array[offset].data += data;
        else    enable_interrupts();
                disable_interrupts();
```

Figure 8: User-level polling and an inlined handler on Alewife.

User-Level Polling: Given inexpensive enabling and disabling of interrupts, such as with the *user-polling* bit on Alewife, we can use the RQ abstraction to implement a mixed model that allows the user to poll for particular messages and allows system and other messages to generate interrupts.

Since the current Alewife system does not support the *user-polling* bit yet, we approximated this behavior using the global flags described in Section 3.2.3. The application turns interrupts off (using *user_atomicity_request*) with a very large timeout. At polling points (Figure 8), the application treats the NI as a remote queue and examines the header of any waiting message. The application then decides whether to process the message with the code within the loop or to enable interrupts (using *user_atomicity_release*) and allow the message to generate an interrupt. This is similar to the pseudo-polling from Spertus *et al.* [SPE+93], except that most messages are processed via true polling. Once all message interrupts have been processed, control returns to the polling loop and interrupts are disabled (using *user_atomicity_request*). Note that enabling interrupts resets the timer for atomicity timeouts. With the new polling-bit mechanism, users will be able to dequeue packets individually, which has added benefits covered in Section 4.

Although this implementation only approximates the *user-polling* bit design, it achieves most of the performance and atomicity advantages. It also retains some of the protection benefits of the timeout mechanism. Even though the timeout is set to an extremely large value, the eventual timeout will interrupt a deadlocked program, freeing machine resources or allowing it to be debugged.

3.2.4 Block Transfers

Integration of RQ and DMA in Alewife is a natural consequence of integration within the user-level network interface. As illustrated by Figure 6, message description can include optional blocks of memory to be appended to the message interface via DMA. At the time that a message is launched, a optional *transmission completion interrupt* can be requested to signal that data being transferred through DMA has been committed to the network.

At the receiver, portions of an incoming packet may be stored to memory with DMA. As with message transmission, the user may request a *storeback completion interrupt* to signal that input DMA has finished. This interface illustrates an important advantage of implementing portions of the RQ model directly with the NI: a handler can determine where data should be stored before initiating DMA. This avoids the multiple copies that occur when messages are copied to memory before the user-level handler executes.

3.2.5 Performance

Table 3 summarizes the latencies to send and receive a null user-level active message with protection but without the proposed

Message Send	Cost (cycles)
Descriptor construction	6
Launch	1
Total	**7**
Message Reception (from header)	**Cost (cycles)**
Post of interrupt & pipeline flush	6
Register allocation & first dispatch	16
Atomicity check	2
Timer setup	11
Second dispatch	15
Stub execution	5
Return to supervisor	5
Restore system timer	14
Return from trap	21
Total	**95**

Table 3: Cycle count to send and receive a null user-level active message, with protection but without the proposed hardware fixes on Alewife

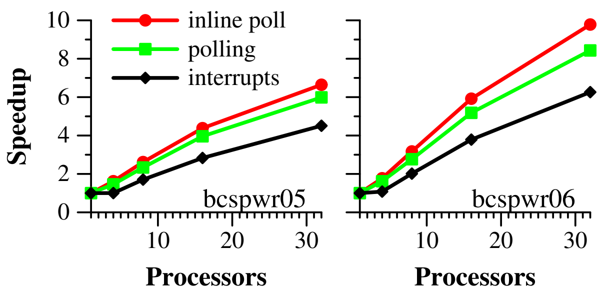

Figure 9: Performance of *interrupts, user-level polling without inlining,* and *user-level polling with inlining* on sparse triangular solution of incomplete factors of Harwell-Boeing power matrices *bcspwr05* and *bcspwr06* on Alewife.

hardware fixes on Alewife.

As illustrated in Figure 4, handlers may be specifically optimized for an application when using RQ. Figure 9 shows the performance of *interrupts, polling without inlining*, and *polling with inlining* on Alewife on sparse triangular solution of incomplete factors of Harwell-Boeing power matrices *bcspwr05* and *bcspwr06*. Speedups are relative to an optimized sequential code running on a single Alewife node. The data points for one processor are also for this sequential code. Although Alewife supports extremely fast message interrupts, both polling implementations outperform the interrupt-driven implementation, by as much as 30 percent. Note that the interrupt code is running on a fast kernel with no protection. The polling code runs on a protected kernel and will gain an even bigger advantage with the new CMMU.

We can see the advantage of code optimizations enabled by polling. The *polling with inlining* code includes inlining of the handler and efficient conversion between single word message data and double word handler arguments. By comparing the two polling codes, we can see that polling enables code optimizations that improve performance by as much as 6 percent in this application.

In summary, our preliminary user-level polling implementation on Alewife demonstrates that we can achieve superior applications performance while retaining the protection and flexibility of interrupts. The revised CMMU will increase the advantages of polling.

3.3 Cray T3D

The Cray T3D is a multiprocessor with physically distributed but globally addressed memory. Each node of the T3D is a 64-bit, 150-MHz DEC Alpha microprocessor with an 8KB data cache. The processors are interconnected by a 3D torus network.

The memory interface between the Alpha and the local memory on each node uses custom circuitry that integrates local memory into the global address space. Five bits of the virtual address are

```
enqueue(proc, arg1, arg2) {
    /* set up annex for RQ remote pointers */
    RQ_head = (int*) map_dtb_address(pe,f_inc_reg);
    RQ_addr = (Word*) map_dtb_address(pe,RQ);
    if (*RQ_head < Q_SIZE ) {
        /* write data to RQ */
        RQ_addr[*RQ_head].func  = handler;
        RQ_addr[*RQ_head].word1 = arg1;
        RQ_addr[*RQ_head].word2 = arg2;
        /* presence word completes operation */
        RQ_addr[*RQ_head].word3 = VALID;
        return(TRUE);
    } else return(FALSE);

dequeue(ptr1, ptr2)
    if (RQ_addr[tail].word3 == VALID ) {
        *ptr1 = RQ_addr[tail].word1;
        *ptr2 = RQ_addr[tail].word2;
        /* clear element */
        RQ_addr[tail].word3 = INVALID;
        /* advance the queue */
        if (tail < Q_SIZE-1) tail++ else reset_Q();
        return(TRUE);
    } else return(FALSE);

/* inlined poll */
while (dequeue(&offset, &data))
    array[offset].outstanding++;
    array[offset].data += data;
```

Figure 10: RQ and inlined polling on the T3D

used to index into a look-up table, called the *annex*, that contains processor numbers and opcodes that define remote functions. This allows each processor to read or write any other processor's memory directly without interrupting that node. There is no global cache coherence: it is the user's responsibility to keep the caches coherent, but the support circuitry allows remote data entering a processor's local memory to invalidate the corresponding cache line.

The T3D also supports several special functions that are useful for distributed-memory programming. Each processor has a fetch-and-increment register that can be accessed by other processors. A special *swaperand* register on each processor can be used to perform atomic operations on other processors' memory.

3.3.1 Implementing Active Messages on top of RQ

We use the fetch-with-increment operation on the T3D to implement RQ with queues in user memory. Active messages with interrupts is not possible, but we can build active messages with polling on top of RQ. This provides the first effective fine-grain communication on the T3D, other than remote writes, and enables good performance on fine-grain applications like sparse-matrix solution and Split-C.

Figure 4 outlines code to implement RQ and inlined polling. Each processor dedicates a globally accessible region in memory to act as the queue. Due to the T3D's remote-write facility, the RQ memory locations can be written directly by other processors. A queue counter is associated with each queue; incrementing the queue counter must be atomic. This atomicity is achieved by using the fetch-and-increment register on each node of the T3D to implement the counter. Since there is only one fetch-and-increment register per node, our implementation allows only one queue per processor, but this is sufficient since each application gets dedicated hardware via partitions.

To send a message, the sender obtains a pointer into the RQ by reading the fetch-and-increment register on the receiver. The process of reading this register atomically increments it, thus ensuring that two processors do not write into the same queue entry. After obtaining the RQ counter, the sender forms a message, which is done by storing data into a local cache-line and then directly writing the 32-byte cache-line message into the remote queue (in the position reserved by the RQ counter). Writing into remote memory does not interrupt the processor.

Messages are explicitly dequeued at the receiver, which directly corresponds to polling. The receiver looks at the RQ counter, which is just the local fetch-and-increment register, and reads all new messages from the queue. We note that the sender increments the queue counter before actually sending the data. This means that the receiver might mistakenly try to pick up data from the RQ before it has actually arrived. One work-around is to interpret a predefined area of each incoming message (cache-line) as a presence flag, as we did on the Paragon. Note that this polling is done on a cached memory location corresponding to the position in the queue where the presence flag is expected to be set. To maintain consistency between cache and memory, we flush the cache line corresponding to the incoming global address.

3.3.2 Deadlock Avoidance

Deadlock is not an issue for RQ on the T3D since all communication is via remote reads and writes that are handled asynchronously by the DMA engine.

3.3.3 Flow Control and Queue Overflow

However, flow control is necessary to prevent overflow of our fixed-size queues. Senders read the RQ counter and verify that it is less than the queue size. If not, then the sender keeps polling the counter until it is reset by the receiver. No messages are sent until the value obtained is less than the queue size. This means that if there is insufficient polling, the network does not get congested. Instead, some bandwidth is wasted on accessing the queue counter and the overhead is increased by the round-trip of a fetch-and-add. The data simply remains at the sender and thus affects neither the links nor the intermediate routers. In addition, this scheme does not require the queue size to scale with the number of processors and the depth of the network as in the Paragon implementation.

3.3.4 Block Transfers

It is relatively simple to integrate RQ handlers with the T3D's DMA-based block transfers. Since the network provides in-order delivery, we can simply enqueue a message after the DMA has started at the sender. When the receiver handles the message, it knows that the block transfer has completed and can act appropriately. For example, it is trivial to count incoming data, which allows the receiver to know when a data-transfer phase has ended.[4]

3.3.5 Integration of Interrupts

It should be noted that the T3D has hardware for messages with interrupts. However, it is inefficient for fine-grain applications since message arrival generates an interrupt and requires a context switch into the operating system. The cost of receiving a message ranges from 5 µsec to 58 µsec depending on the frequency of message arrival and the mode of operation being used [Arp+95]. If we actually need active messages with interrupts (or RQ with interrupts), then the kernel must support some form of upcall to a user-level interrupt routine. In fact, the Cray T3E will support message passing with user-level interrupts, but given the overhead most applications will still use explicit dequeuing (as on the CM-5).

3.3.6 Performance

Table 4 gives the cost breakdowns of active messages on top of RQ. Figure 11 shows the performance of the sparse-matrix application. our fine-grain application with RQ on the T3D. The speedups, although small due to the fine granularity, are consistant with the Alewife implementation, despite the latter's integrated support for communication.

4: This important operation is not possible using only the barrier hardware, since the barrier indicates that all nodes have finished sending, not that all of the data has arrived. With counting, we can perform a barrier after all of the local data has *arrived*.

`send_am()`	Cost (µs)
Fetch & Increment	0.77
Update annex pointer	4.50
Flow control checks	0.18
Cache write + memory barrier	0.81
Total (preset/non-preset annex)	**1.76/6.26 µs**
`poll()`	**Cost (µs)**
Read F&I register	0.53
Handler invocation overhead	0.44
Flow control checks	0.00
Check packet's presence bit	0.42
Read data, update queue pointer	0.59
Total	**1.98 µs**
Round Trip Time	**Cost (µs)**
Sender: `send_am()`	1.76/6.26
Network latency	1.05
Receiver: `poll()`	1.54
Receiver: `send_am()`	1.76/6.26
Network latency	1.05
Sender: `poll()`	1.98
Total (preset/non-preset annex)	**9.12/18.1 µs**

Table 4: Cost breakdowns for active messages on top of RQ on the Cray T3D. For the round-trip time, the receiver's `poll` cost excludes the handler overhead (0.44 µs), since it is reflected in the subsequent `send_am` call. For repetitive communication with a limited number of receivers, the annex need not be set for each `send_am` call, which reduces the overhead from 6.26 µs to only 1.76 µs.

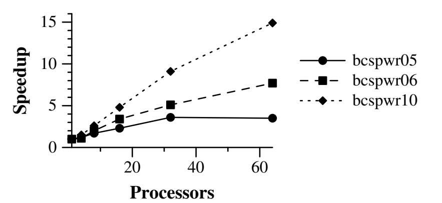

Figure 11: T3D Peformance on small sparse matrices.

4 RQ and High-Level Communication

The Remote Queue model has two facets that lead to better performance when used as a substrate for high-level communication such as cyclic shifts or transpose patterns. First, because we a know a great deal about the incoming data, we can optimize the movement of data. Second, the ability to limit reception, unlike interrupt-based active messages, can lead to reduced overhead and more even communication rates among the senders.

There are several issues that affect the performance of block transfers. First, some coprocessors support DMA and some do not. We exploit DMA on Alewife, Paragon and the T3D, but on the CM-5 block transfers must be built out of small packets. For the CM-5 and for strided transfers on all four platforms, we build block transfers on top of RQ. We return to strided transfers in Section 4.1.

Figure 12 shows the impact of RQ on the CM-5 for 64-node transpose patterns with 4K blocks. Transpose, which is one of the more difficult patterns, is implemented as a sequence of permutations separated by barriers to maximize performance [BK94].

The first bar shows the performance of transpose using block transfers built on top of active messages. The first optimization is to exploit our knowledge about incoming data. Instead of invoking handlers for each incoming packet, we dequeue the packets in a tight loop. This eliminates invocation overhead and enables a new optimization: we can keep a pointer to the target memory as a hint

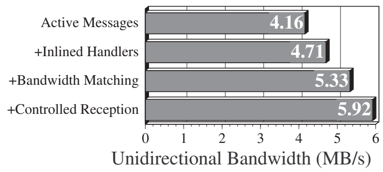

Figure 12: The impact of RQ optimizations on transpose patterns with 4K blocks. First, we dequeue the incoming data in a tight loop. Next, we use bandwidth matching to reduce congestion (which does not depend on RQ). Finally, we dequeue exactly one packet between injections. Overall, we achieve a 42% improvement.

for the next packet. The hint eliminates the overhead involved in converting segment-offset information into actual addresses. In total, we get a 13% speedup, which is consistent with the results in [LC94]. This optimization previously existed in both Strata and CMMD; the key is to realize that this is just an exploitation of RQ.

The second optimization involves RQ only indirectly. Almost exclusively, multiprocessor networks depend on *backpressure* for flow control. A blocked receiver thus backs up traffic all the way to the sender by incrementally blocking the routers along that path. The primary advantages of backpressure are simplicity and the ability to prevent buffer overflow. Unfortunately, backpressure does not affect the sender until the path between the sender and receiver is completely congested. In the presence of adaptive routing, *all* of the paths to the receiver must be congested before the sender detects any problem.[5] By the time the sender stops injection, the network is severely congested, which hinders all of the nodes.

The solution in wide-area networks is to use window flow control, which limits the maximum number, w, of outstanding packets to any specific receiver. This has the advantage that the sender will inject at most w packets without some feedback from the receiver. Thus, the network remains clear for the rest of the nodes. Unfortunately, window flow control can double the software overhead for active-message based communication [KC94].[6]

An alternative is *rate-based* flow control, which explicitly limits the injection rate rather then the total number of outstanding packets. *Bandwidth matching* [BK94] statically limits the injection rate to match that of the bottleneck, which is normally the receiver. Assuming an attentive receiver, which we can ensure for bulk communication patterns, bandwidth matching leads to optimal flow control: the network contains the minimum number of packets required to reach the full bandwidth of every receiver. Note that bandwidth matching adds zero overhead; it simply limits the sender to the bottleneck rate artificially. Although bandwidth matching has little impact on individual block transfers, it leads to an additional 13% improvement for transpose, as shown by the third bar in Figure 12. Bandwidth matching also works if the bottleneck is the bisection or in the presence of interrupts instead of dequeues.

RQ enables a third optimization. It is not enough to limit the injection rate of the sender; it also critical to ensure *even progress* of the senders. In the final bar, we exploit the ability to accept exactly one packet to reduce overhead and ensure even progress of the senders. If we use interrupts or poll until the queue is empty, we may delay the *injection* of packets for a long time, since data continues to arrive. In the worst case, sending is delayed until all of the incoming packets have arrived, which nearly doubles the overall time for the transpose. Bandwidth matching helps this by ensuring that the receiver can drain the network faster than the injection rate. This means that the receiver now has the freedom to delay accepting a packet, since bandwidth matching avoids congestion. Thus, for the final optimization, we strictly alternate sending and receiving in the normal case, which ensures fairness and reduces overhead by eliminating most of the failed dequeues. This is worth an additional 11%, and 42% overall compared with active messages. We should note that limited reception without bandwidth matching leads to terrible congestion, achieving half of the bandwidth.

Thus, for transpositions and other forms of cyclic shifts [FOX+88], RQ supports higher throughput for all receivers than was seen for *individual* block transfers using TMC's optimized libraries [KTR93]. Systems with DMA would not use RQ for block transfers, although a combination of DMA and RQ is useful for counting incoming data, which is required by applications such as radix sort.

4.1 Strided Block Transfers

Although DMA is useful for block transfers, it is much harder to exploit for strided block transfers. Instead we build strided transfers using either RQ or a combination of RQ and DMA. Strided transfers are both important and difficult. For example, in iterative PDE applications, the boundaries must be communicated to (logically) nearest neighbors. For a given layout, some of these boundaries will be contiguous and some will require strided transfers.

These transfers are challenging for two reasons. First, because the transfers consist of a sequence of relatively small blocks, the overhead of DMA often makes RQ a better choice. For example, for an array of 100x100x100 doubles, the strided boundary planes contain 10,000 8-byte blocks.

Second, the strided boundaries tend to cover a large range of the address space. In the array example, four of the boundary planes each cover nearly all of the 8MB allocated for the array. This leads to poor cache and virtual-memory performance. In fact, if the system must pin pages for incoming data, we must either pin all of the pages, or pin and unpin pages in the middle of the transfer.

RQ mitigates both of these problems. First, since RQ has much lower overhead for small blocks than DMA, we can build faster strided block transfers using RQ as we did with (regular) block transfers on the CM-5.

Second, since we dequeue packets in a tight optimized loop, we can easily add prefetching for caches lines and/or pages. Just as we exploited knowledge about the next packet to optimize address calculations, we can prefetch the cache line for the next expected block. We can also prefetch pages or pin/unpin pages as we go.[7]

We can extend these ideas from strided block transfers to the *General Datatypes* specified by MPI [WAL94]. These datatypes allow users to define transfers involving very general collections of primitive types such as floats and integers. For example, a user might define a datatype that consists of multiple kinds of strided arrays. Since this type exactly specifies the nature of the incoming packets, we can exploit the same kinds of optimizations. In fact, given our integration of RQ and DMA, we can easily mix DMA and RQ-based transfers depending on the size of each block.

Note to referees: we expect to have data on the impact of cache prefetching but not page prefetching for the final version. Cache prefetching also helps a little for regular block transfers, but not as much due to the sequential access pattern.

5 Related Work

There are many machines that have explicit queues in the network interface or communication coprocessor. Some of these are FLASH [KUS+94], *T [PBGB93], Typhoon [RLW94], Meiko CS-2

5: When endpoint congestion becomes the bottleneck, this becomes a strong argument against adaptive routing. Networks that use adaptive routing only at connection setup avoid this problem; examples include the Metro router [DEH+94] and compressionless routing [KLC94].

6: For unreliable networks, as in most networks of workstations, this overhead is already required, so window flow control may be fine.

7: Unfortunately, pinning and unpinning pages is too slow in current operating systems to make this effective. A nonblocking pin/unpin operation would let us overlap the pins with communication.

[SS95], and IBM SP2 [STU+94]. RQ maps well onto all of these machines. *T provides a fairly direct implementation of RQ.

The importance of user-level handlers remains up for debate, with FLASH arguing that prohibiting such handlers simplifies the coprocessor and allows multiprogramming without gang scheduling. Typhoon allows user-level handlers, but isolates users via strict gang scheduling. The RQ answer is that only system code runs on the coprocessor, but that the system codes supports enqueue and specialized handlers for memory access, atomic memory operations, and barriers, scans and reductions. Enqueue allows unrestricted user-level handlers to run on the main processor, which provides greater flexibility, but less performance than running the handlers on the coprocessor. Specialized handlers mitigate this performance hit substantially.

Optimistic Active Messages [WHJ+95] solve the deadlock problem for user handlers by creating and suspending a thread when a lock or other synchronization event is encountered in the handler. Assuming that the rest of the message is drained from the network, this amounts to adaptively moving the message into a queue in user memory. The polling mechanism that we developed for Alewife have a similar flavor: if the atomicity bit is set we postpone handling the message, but still drain the network.

6 Conclusions

We have demonstrated that a range of applications benefits from polling, even when fast interrupts are available. We have introduced efficient atomicity mechanisms for Alewife and shown how these mechanisms can be used to integrate RQ with interrupts. In summary, RQ has the following advantages:

➤ Clean support for atomicity. Code is atomic by default except for explicit dequeues. We can mix polling with interrupts by selectively using interrupts for system messages or infrequent user messages.

➤ RQ allows us to exploit knowledge, if any, about incoming messages. In particular, we can avoid handler dispatch and integrate reception directly into the computation loop, as was done for the sparse-matrix code.

➤ Explicit control of reception enables smoother flow control. In particular, we can precisely alternate sending and receiving, which leads to optimal performance for complex bulk communication patterns such as transpose.

➤ It is easy to integrate RQ with the rest of the communication system. We mixed RQ with DMA on the Paragon, T3D and Alewife. We also mixed RQ for user messages with selective interrupts for either system messages or unexpected user messages. We also showed that RQ enables new optimizations for complex patterns and strided transfers.

➤ We showed that RQ makes sense for coprocessors. It simplifies multiprogramming by separating the draining of the network from the handling of messages. This allows user-level handlers without restrictions, without deadlock, and without the need for strict gang scheduling.

➤ Finally, the benefits of RQ for coprocessors apply equally well to intelligent host interfaces in networks of workstations, which also require multiprogramming and integration of polling and interrupts.

Acknowledgments: Thanks to David Culler, Mark Hill, Alan Mainwaring, Rich Martin, David Kranz, Beng-Hong Lim, Sramana Mitra, Deborah Wallach, and anonymous referees. Ken MacKenzie and Kirk Johnson provided some of the Alewife measurements.

[ACP95] T. E. Anderson, D. E. Culler and D. A. Patterson. The Case for NOW. To appear in *IEEE Micro*, 1995.

[AGA+93] A. Agarwal *et al.* Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro*, 48–61, June 1993.

[ARP+95] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective, *ISCA '95.*

[BK94] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network, *IPPS '94*.

[CSBS95] F. T. Chong, S. D. Sharma, E. A. Brewer and J. Saltz. Multiprocessor Runtime Support for Irregular DAGs. To appear in *Toward Teraflop Computing and New Grand Challenge Applications*, R. K. Kalia and P. Vashishta, editors. Nova Science Publishers, 1995.

[CDG+93] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. *Supercomputing '93*.

[DEH+94] A. DeHon *et al*. METRO: A Router Architecture for High-Performance Short-Haul Routing Networks. *ISCA '94*.

[FOX+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Volume 1: General Techniques and Regular Problems*. Prentice Hall, 1988.

[HB92] I. S. Duff, R. G. Grimes and J. G. Lewis. *User's Guide for the Harwell-Boeing Sparse Matrix Collection*. CERFACS Technical Report TR/PA/92/86, October 1992.

[INTEL91] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[KA93] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. *International Conference on Supercomputing*, July 1993.

[KC94] V. Karamcheti and A. A. Chien. Software Overhead in Messaging Layers: Where Does the Time Go? *ASPLOS '94.*

[KLC94] J. H. Kim, Z. Liu and A. A. Chien. Compressionless Routing: A Framework for Adaptive and Fault-Tolerant Routing. *ISCA '94*.

[KTR93] T. T. Kwan, B. K. Totty, and D. A. Reed. Communication and Computation Performance of the CM-5. *Supercomputing '93*, pages 192–201.

[KUS+94] J. Kuskin *et al*. The Stanford FLASH Multiprocessor. *ISCA '94*.

[LEI+92] C. E. Leiserson *et al*. The Network Architecture of the Connection Machine CM-5. *SPAA '92*, March 21, 1994.

[LC94] L. T. Liu and D. E. Culler. *Measurements of Active Messages Performance on the CM-5*. UCB TR CSD-94-807.

[MMRW93] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A Brief User's Guide. *Proceedings of the Intel Supercomputer Users Group Annual North America Users Conference*, June 1993.

[PBGB93] G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Berkerle. *T: Integrated Building Blocks for Parallel Computing. *Supercomputing '93*.

[PIE94] P. Pierce. The NX Message Passing Interface. *Parallel Computing*, Vol. 20, 463–480, April 1994.

[RLW94] S. K. Reinhart, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. *ISCA '94*.

[SEI94] C. Seitz. Myrinet — A Gigabit-per-Second Local-Area Network, *Proceedings of Hot Interconnects II*, August 1994.

[SPE+93] E. Spertus *et al.* Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. *ISCA '93*.

[SS95] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. IPPS '95.

[STU+94] C. B. Stunkel *et al.* The SP2 Communication Subsystem. On-line as: http://ibm.tc.cornell.edu/ibm/pps/doc/css/css.ps.

[vE+92] T. von Eicken *et al*. Active Messages: A Mechanism for Integrated Communication and Computation. *ISCA '92*.

[WAL94] D. W. Walker. The Design of a Standard Message Passing Interface for Distributed-Memory Concurrent Computers. *Parallel Computing*, Vol. 20, 657–673, April 1994.

[WHJ+95] D. A. Wallach, W. C. Hsieh, and K. L. Johnson, M. F. Kaashoek, W. E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. PPoPP '95.