Evaluating Design Alternatives for Reliable Communication on High-Speed Networks

Raoul A. F. Bhoedjang, Kees Verstoep, Tim Rühl, Henri E. Bal, and Rutger F. H. Hofman

Dept. of Computer Science
Cornell University
Ithaca, NY, USA
raoul@cs.cornell.edu

Dept. of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
{versto, bal, rutger}@cs.vu.nl

Data Distilleries, Inc.
Amsterdam, The Netherlands
t.ruhl@datadistilleries.com

ABSTRACT

We systematically evaluate the performance of five implementations of a single, user-level communication interface. Each implementation makes different architectural assumptions about the reliability of the network hardware and the capabilities of the network interface. The implementations differ accordingly in their division of protocol tasks between host software, network-interface firmware, and network hardware. Using microbenchmarks, parallel-programming systems, and parallel applications, we assess the performance impact of different protocol decompositions. We show how moving protocol tasks to a relatively slow network interface yields both performance advantages and disadvantages, depending on the characteristics of the application and the underlying parallel-programming system. In particular, we show that a communication system that assumes highly reliable network hardware and that uses network-interface support to process multicast traffic performs best for all applications.

1. INTRODUCTION

Efficient support for reliable point-to-point and multicast communication is a fundamental component of parallel-programming systems (PPSs) and their applications. While research on user-level communication architectures has resulted in efficient communication systems [11, 22, 24, 26], these systems take widely different approaches to implementing reliable communication [6]:

- Due to the high reliability of many modern networks, several research systems do not implement traditional, retransmission-based reliability protocols [21, 22, 24]. Other systems do use retransmission, but differ in whether they implement it on the host or on the network interface (NI) [11].
- On scalable, switched networks, multicasting is usually implemented by means of spanning-tree protocols that forward multicast data in software. Most systems implement spanning-tree multicasts on top of message-based point-to-point primitives. Several recent systems, however, let the NI forward multicast traffic, because this results in fewer data transfers and interrupts on the critical path from sender to receivers [5, 12, 14, 25].

To appear in *ASPLOS 2000* Cambridge, MA, USA

	Multicast forwarding	
Retransmission	By host (M_h)	By NI (M_{ni})
None (RX_{no})	$RX_{no}M_h$	$RX_{no}M_{ni}$
By NI (RX_{ni})	$RX_{ni}M_h$	$RX_{ni}M_{ni}$
By host (RX_h)	RX_hM_h	Not implemented

Table 1: Five LCI implementations.

A systematic evaluation of these design choices is difficult, because existing systems provide different programming interfaces and functionality. Existing studies often use only microbenchmarks and ignore multicast [1]. Moreover, the impact of the design choices at the level of PPSs and applications has hardly been investigated.

In this paper, we use five implementations of a *single*, Low-level Communication Interface (LCI) to assess the impact of different designs on parallel applications. Since we use a single communication interface, we do not have to modify applications or their PPSs to conform to different communication paradigms.

Each LCI implementation uses one of three reliability schemes and one of two multicast schemes (see Table 1). These schemes make different assumptions about the network (reliable or not) and the capabilities of the network interface (intelligent or dumb). Accordingly, the LCI implementations divide protocol tasks differently between the host software, the NI firmware, and the network hardware. Each decomposition makes a different tradeoff between functionality, host occupancy, NI occupancy, end-to-end latency, and throughput. We used microbenchmarks and applications to tune and optimize all LCI implementations and to measure the impact of their different work decompositions. An interesting outcome of these measurements is that there is no strict fastest-to-slowest ordering of decompositions for all applications.

Our specific contributions are as follows:

- We show that differences in protocol decompositions have a significant application-level performance impact, in spite of the interposition of parallel-programming systems with relatively large overheads. The PPSs used in this paper add 25–124% to LCI-level latencies. One would expect these overheads to dominate application-level performance.
- We show how NI support for reliability and multicast can yield both performance advantages and disadvantages. In

particular, we show that an implementation that assumes reliable network hardware and that uses NI support to implement flow control and multicast forwarding always performs best. With other protocol decompositions, applications perform up to 45% slower.

- We correlate application-level performance, differences between the LCI implementations, the communication styles of PPSs, and properties of applications. In particular, we show that the importance of communication parameters such as send and receive overhead depends on PPS-specific communication styles.

The paper proceeds as follows. Section 2 describes LCI and Section 3 describes our implementations of this interface. Section 4 compares the implementations using microbenchmarks. Section 5 summarizes the performance of the PPSs used in our evaluation. Section 6 analyzes application performance. Section 7 discusses related work and Section 8 concludes.

2. COMMUNICATION INTERFACE (LCI)

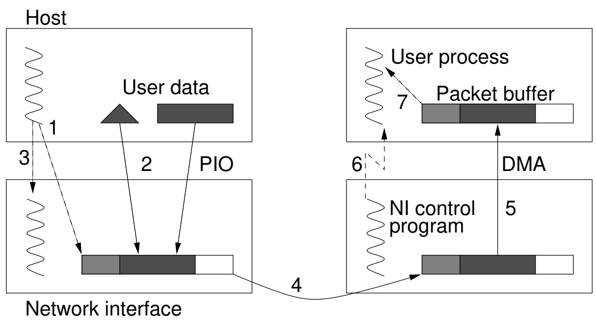

Figure 1: LCI's packet-based programming interface.

LCI provides reliable, packet-based point-to-point and multicast communication. Packets are delivered in FIFO order and can be received using polling or interrupts. For synchronization, LCI provides an atomic, remote fetch-and-add (F&A) primitive.

Figure 1 illustrates the programming interface. To send data, an LCI client allocates a send buffer in NI memory (step 1) and uses programmed I/O (PIO) to copy data into this packet buffer (2). Next, the client hands the packet to a point-to-point or multicast routine which passes a send request to the NI (3). The NI transmits the packet to the destination NI (4), which copies the packet to pinned host memory, using DMA (5). If network interrupts are enabled, the NI generates an interrupt (6). The host passes the packet to a client-supplied upcall routine (7), which processes it and then releases it or queues it for further processing.

3. IMPLEMENTATION OVERVIEW

All LCI implementations run on the Myrinet cluster described in Section 4. Each implementation consists of a reliability scheme and a multicast scheme (see Table 1). The different schemes divide protocol tasks differently between the host and Myrinet's programmable NI. The reliability schemes are *no retransmission* (RX_{no}), *host-level retransmission* (RX_h), and *NI-level retransmission* (RX_{ni}). The multicast schemes are *host-level multicast* (M_h) and *NI-level multicast* (M_{ni}).

3.1 No Retransmission (RX_{no})

RX_{no} assumes reliable network hardware with FIFO communication paths. RX_{no} therefore never retransmits and needs neither sender-side packet buffering nor timer management. Also, with FIFO communication paths, there is no need to maintain sequence numbers. The implementation fails if one of the assumptions is violated (e.g. if the hardware drops or corrupts a network packet).

To avoid buffer overflow, RX_{no} implements credit-based sliding-window flow control between each pair of NIs. Each NI reserves a fixed number W of receive buffers for each sender. Each NI transmits a packet only if it knows that the receiving NI has free buffers. If a packet cannot be transmitted due to a closed send window, the NI queues the packet on a per-destination *blocked-sends queue*. Blocked packets are dequeued and transmitted during acknowledgement processing.

RX_{no} reuses send buffers after their transmission and therefore needs only a few send buffers. An occupied NI receive buffer is released when the packet contained in it has been copied to host memory. The number of newly released buffers is piggybacked on each packet that flows back to the sender. If there is no return traffic, then the receiving NI sends an explicit half-window acknowledgement after releasing $W/2$ buffers.

3.2 Host-Level Retransmission (RX_h)

RX_h implements reliability in a largely traditional way, on the host processor. RX_h assumes unreliable network hardware and recovers from lost, corrupted, and dropped packets by means of timeouts, retransmissions, and hardware-supported CRC checks. RX_h differs from a traditional protocol, because it uses NI memory to buffer packets for retransmission. Since the Myrinet hardware can transmit only packets that reside in NI memory, all LCI implementations must copy data from host memory to NI memory. RX_h uses the packet in NI memory for retransmission and therefore does not make more memory copies than RX_{no} (unlike many traditional implementations).

RX_h uses a similar sliding-window protocol as RX_{no}, but runs it on the host. After transmitting a packet, the sender starts a retransmission timer. When this timer expires, unacknowledged packets are retransmitted. This go-back-N protocol is efficient if packets are rarely dropped or corrupted.

Each packet carries a sequence number which is used to detect lost, duplicate, and out-of-order packets. Receivers drop out-of-sequence packets. NIs drop packets with CRC errors and packets that cannot be stored due to a shortage of NI receive buffers.

In retransmitting protocols, send packets cannot be reused until they have been acknowledged. Given the small size of NI memories, a sender that communicates with many receivers may run out of send buffers before acknowledgements flow back. One solution is to acknowledge each data packet, but this increases network, NI, and host occupancy. Instead, RX_h uses piggybacked and half-window acknowledgements, just like RX_{no}. In addition, however, each receiver maintains an acknowledgement timer per sender S. The timer for S is started when a data packet from S arrives and the timer is not yet running. The timer is canceled when an acknowledgement, possibly piggybacked, travels back to S. If the timer expires, the receiver sends an explicit *delayed* acknowledgement.

3.3 Interface-Level Retransmission (RX_{ni})

RX_{ni} implements almost the same reliability protocol as RX_h but runs it between NIs rather than between host processes. Sliding-window and timer management, acknowledgement processing, etc., are all performed on the NI. This increases NI occupancy, but reduces host overhead.

	Reliability	Mcast	Fine-grain timer	F&A
$RX_{no}M_{ni}$	NI	NI	—	NI
$RX_{no}M_h$	NI	host	—	NI
$RX_{ni}M_{ni}$	NI	NI	NI	NI
$RX_{ni}M_h$	NI	host	NI	NI
RX_hM_h	host	host	NI+host	host

Table 2: Division of work in the LCI implementations.

Since NI memories are small compared to host memory, RX_{ni} is more likely to drop packets than RX_h. NIs therefore keep track of available receive buffer space. When a packet must be dropped due to a buffer space shortage, the NI requests retransmissions from the sender as soon as buffers become available again. Without this mechanism, packets would not be retransmitted until the sender's timer expired.

3.4 Host-Level Forwarding (M_h)

Multicasting on scalable, switched networks is usually implemented by means of spanning-tree protocols that forward multicast data. M_h uses this scheme and organizes the receivers of a multicast in a binary tree, with the sender as its root. The sender transmits multicast packets to each of its children. Each receiving NI passes incoming multicast packets to its host, which reinjects them to forward them to the next level in the tree.

M_h performs two important optimizations. First, instead of making a separate host-to-NI copy for each forwarding destination, the host copies each packet only once and creates multiple transmission requests for the same packet. Second, M_h forwards individual multicast packets rather than entire multicast messages. In systems that forward messages, no tree node starts to forward a multicast message until it has received and reassembled the entire message. This strategy allows multicasting to be implemented on top of message-based point-to-point primitives, but greatly reduces multicast throughput.

3.5 Interface-Level Forwarding (M_{ni})

In M_{ni}, receiving NIs recognize multicast packets, copy them to host memory, look up their forwarding destinations, and forward them without host intervention. This NI-level forwarding has three advantages. First, the host does not reinject multicast packets into the network, which saves a host-to-NI data transfer at each internal node of the multicast tree. Second, the critical sender-receiver path includes no host-NI interactions, which potentially reduces multicast latency. Third, forwarding takes place even if the host does not poll the network in a timely manner. With host-level forwarding, the NI can raise an interrupt to force the host to forward a packet. On our experimental platform, however, interrupts increase receive overhead by 31 μs; this is more than the roundtrip latency of a small message.

3.6 Complete Implementations

The reliability and multicast schemes can be combined in six ways; we implemented five combinations on a Myrinet [8] cluster (see Table 1). RX_hM_{ni} has not been implemented, because it requires sharing the sequence-number administration between the host and the NI, which is inefficient. The implementations are mostly user-level; only interrupts are vectored through the kernel. Multiprogramming and protection have not been implemented; the user process has direct access to the NI.

We use Myrinet's programmable NI to divide protocol tasks between the host and the NI in different ways. Table 2 summarizes the division of work in each implementation. Reliability and multicast forwarding were discussed above. $RX_{ni}M_{ni}$ and $RX_{ni}M_h$ use the NI's on-board timer to implement retransmission and acknowledgement timers. RX_hM_h uses an NI-supported variant of soft timers [2] to implement fine-grain timers. As with soft timers, the host polls the clock (in our case, the CPU timestamp counter) instead of relying only on coarse-grain OS timer signals. Unlike soft timers, which use OS timers as a backup mechanism against infrequent polling, RX_hM_h lets the NI generate periodic clock interrupts. Due to its higher clock resolution (0.5 μs versus 10 ms), the NI is a more precise backup than the OS. The NI also avoids generating interrupts when no timers are running or when the host recently polled the clock.

A fetch-and-add on an F&A variable is implemented using an RPC to the host or NI that stores the variable. All implementations except RX_hM_h store this variable in NI memory and perform the operation on the NI. RX_hM_h stores F&A variables in host memory and handles F&A requests on the host. This means that an F&A request can generate an interrupt if the host process is not polling at the time the request arrives.

All implementations transmit data in variable-length packets and store these packets in fixed-size buffers. The default buffer size is 1 Kbyte, but we also present measurements for configurations that use 2 Kbyte buffers. In each implementation, the host maintains a ring of free receive buffers. Each buffer contains a full/empty flag that can be polled by the host processor. The NI uses DMA to fill these buffers with incoming packets. All implementations favor polling and use an NI-level software polling watchdog [19] to delay network interrupts for at least 70 μs.

An important difference between the reliability schemes is that RX_{ni} and RX_h allow NI receive buffers to be shared by all sending NIs. An NI that runs out of receive buffers is allowed to drop incoming packets, because the senders will retransmit. With RX_{no}, in contrast, an NI must not drop packets. Each NI therefore allocates a full window of receive buffers to each potential sender. While this is wasteful, the small bandwidth-delay product of RX_{no} ($\approx$ 1.5 Kbyte) allows small window sizes.

Multicast reliability is obtained by building on the point-to-point reliability schemes between nodes in the multicast tree. In all implementations, multicast acknowledgements are merged with point-to-point acknowledgements and flow back along the multicast tree to prevent acknowledgement implosions.

Store-and-forward multicast protocols introduce the danger of deadlock. $RX_{no}M_{ni}$, $RX_{ni}M_{ni}$, and RX_hM_h can prevent such deadlocks by reserving enough receive buffers. Unfortunately, this destroys one of the advantages of the retransmission-based schemes: better utilization of receive buffers. $RX_{no}M_h$ and $RX_{ni}M_h$ would require additional flow control at the forwarding level to guarantee deadlock freedom. We have not implemented this, because our applications have enough internal flow control to prevent deadlocks.

4. MICROBENCHMARKS

We performed measurements on up to 64 nodes interconnected by a Myrinet [8] system area network. All benchmarks discussed in this section and in Section 5 receive all messages through polling.

Each network node contains a 200 MHz Intel Pentium Pro processor with three on-chip caches (8 Kbyte L1 D cache, 8 Kbyte L1 I cache, and 256 Kbyte L2 I+D cache) and 128 Mbyte of DRAM. The I/O bus is a 33 MHz, 32-bit PCI bus. A Myrinet NI is attached to the I/O bus.

Myrinet's programmable NI has a custom RISC processor, 1 Mbyte of SRAM, and three DMA engines. The processor, a 33 MHz LANai4.1, runs an order of magnitude slower than the 200 MHz,

Parameter	$RX_{no}M_{ni}$, $RX_{no}M_h$		RX_hM_h, $RX_{ni}M_{ni}$, $RX_{ni}M_h$	
Max. packet size (bytes)	1024	2048	1024	2048
NI send pool size (buffers)	128	64	256	128
NI recv. pool size (buffers)	512	256	384	192
Host recv. pool size (buffers)	4096	4096	4096	4096
Send window size (packets)	8	4	8	8
Netw. interrupt delay (μs)	70	70	70	70
Timer granularity (μs)	N/A	N/A	5000	5000

Table 3: Configuration parameters.

	RX_{no}	RX_{ni}	RX_h
Send overhead (o_s)	1.5	1.5	2.4
Receive overhead (o_r)	2.5	2.4	6.0
Gap (g)	6.6	9.9	8.3
Latency (L)	6.2	8.5	5.9
End-to-end latency ($o_s + o_r + L$)	10.2	12.4	14.3
F&A latency	20	24	34

Table 4: LogP values and F&A latencies (in microseconds).

superscalar Pentium Pro. The NI's memory holds the code and the data for the LCI control program. All incoming and outgoing network packets must be staged through this memory. One DMA engine transfers data between host memory and NI memory. The two other DMA engines transfer packets from NI memory to the network and vice versa.

The NIs are connected via a 3D grid of 8-port switches and full-duplex 1.28 Gbit/s links. Network packets are cut-through-routed. The switch delay is approximately 100 ns and the maximum distance between two NIs is 10 hops, so the total switch delay is at most 1 μs. Myrinet provides no hardware support for multicast.

Its hardware flow-control protocol and low error rate make Myrinet highly reliable. No packets are dropped if all NIs agree on a deadlock-free routing scheme and remove incoming packets in a timely manner. NI firmware and host software, however, may still decide to drop packets due to buffer shortages.

We performed most measurements discussed in this section and in Sections 5 and 6 on two versions of each LCI implementation: a version with a packet size of 1 Kbyte and a version with a packet size of 2 Kbyte. Table 3 shows all parameter settings. The non-retransmitting implementations (columns 2 and 3) require more NI receive buffers than the retransmitting implementations, because they dedicate receive buffers to individual senders. The window size is dictated by the size of the NI receive buffer pool and the number of processors (64). Since the NI receive buffer pool becomes smaller when the packet is increased to 2 Kbyte, the send window also becomes smaller (4 packets). The retransmitting implementations (columns 4 and 5) require more NI send buffers, because they cannot release send buffers until an acknowledgement arrives. These implementations always use the same window size (8 packets).

4.1 Unicast Performance

Table 4 shows the values of the LogP [10] parameters, the end-to-end latency, and the fetch-and-add latency for $RX_{no}M_{ni}$, $RX_{ni}M_{ni}$, and RX_hM_h. Since multicast forwarding plays no role in these measurements, the table and this section refer to these implementations as RX_{no}, RX_{ni}, and RX_h, respectively.

Since RX_{no} and RX_{ni} perform the same work on the host, they have identical send overhead (o_s) and almost identical receive overhead (o_r). RX_{ni}, however, runs a retransmission protocol on the NI, which is reflected in its larger small-message bottleneck, or gap, and latency ($g = 9.9$ μs versus $g = 6.6$ μs and $L = 8.5$ μs versus $L = 6.2$ μs). RX_h runs a similar protocol on the host and therefore has larger send and receive overheads than RX_{no} and RX_{ni}.

RX_{no} has the best end-to-end latency. As expected, the retransmission support in RX_{ni} and RX_h increases end-to-end latency due to timer management and sender-side buffering. Host-level retransmission increases send and receive overhead (o_s and o_r), while NI-level retransmission increases NI occupancy (g) and latency (L). RX_{ni} increases the end-to-end latency less than RX_h. In RX_{ni} the host and the NI can operate in parallel, because the NI has its own copy of each incoming packet's header. In RX_h, the host-level LCI library must complete packet processing before passing the packet to the application.

RX_{no} also has the fastest F&A implementation (20 μs). RX_h is the slowest implementation (34 μs); recall that RX_h handles F&A requests on the host processor.

Figure 2 shows one-way throughput for both packet-size configurations (1 Kbyte and 2 Kbyte, see Table 3), with and without receiver-side copying. The sawtooth shape of the curves is due to the fragmentation of messages larger than the packet size. Without receiver-side copying, and with 1 Kbyte packets, the figure shows clear differences in throughput between the implementations. Performing retransmission administration on the host (RX_h) increases the per-packet costs and reduces throughput. When the retransmission work is moved from the host to the NI (RX_{ni}), throughput is reduced further. The NI performs the same work as the host in RX_h, but more slowly.

Increasing the packet size to 2 Kbyte increases the peak throughput for all implementations. RX_{ni} and RX_{no} now attain the same peak throughput (76 Mbyte/s). With larger packets, the NI is no longer the throughput bottleneck. Since RX_h performs more work on the host, it benefits less from the packet-size increase than RX_{ni} and RX_{no}.

All LCI clients copy incoming packets during message reassembly. Figure 2 shows that this copying reduces throughput, especially when the destination buffer does not fit into the L2 cache. Due to the Pentium Pro's low *memcpy*() speed for large buffers (52 Mbyte/s), the copying stage becomes a bottleneck. With copying, the differences between the implementations are smaller, especially for large messages. This is partly due to the use of the NI memory as 'retransmission memory,' which eliminates a copy at the sending side.

4.2 Broadcast Performance

Figure 3 shows broadcast latency on 64 processors. We define broadcast latency as the time it takes to reach the last receiver. The latency of an empty message ranges from 60 μs for $RX_{no}M_{ni}$ to 112 μs for $RX_{ni}M_h$. The M_h implementations (the top three curves) perform worse than the M_{ni} implementations because they add two data copies per internal tree node to the critical path, from the NI to the host and back. Using a larger packet size has no effect on the broadcast latency, so we show only the results for a packet size of 1 Kbyte.

Figure 4 shows broadcast throughput on 64 processors, for 1 Kbyte packets and 2 Kbyte packets and with receiver-side copying. We define throughput as the sender's outgoing data rate. Here, interface-level forwarding does not always yield the best results: with 1 Kbyte packets and for message sizes up to 128 Kbyte, for example, RX_hM_h achieves higher throughput than $RX_{ni}M_{ni}$. $RX_{ni}M_{ni}$ cannot hide all its buffer and timer management behind DMA transfers. In

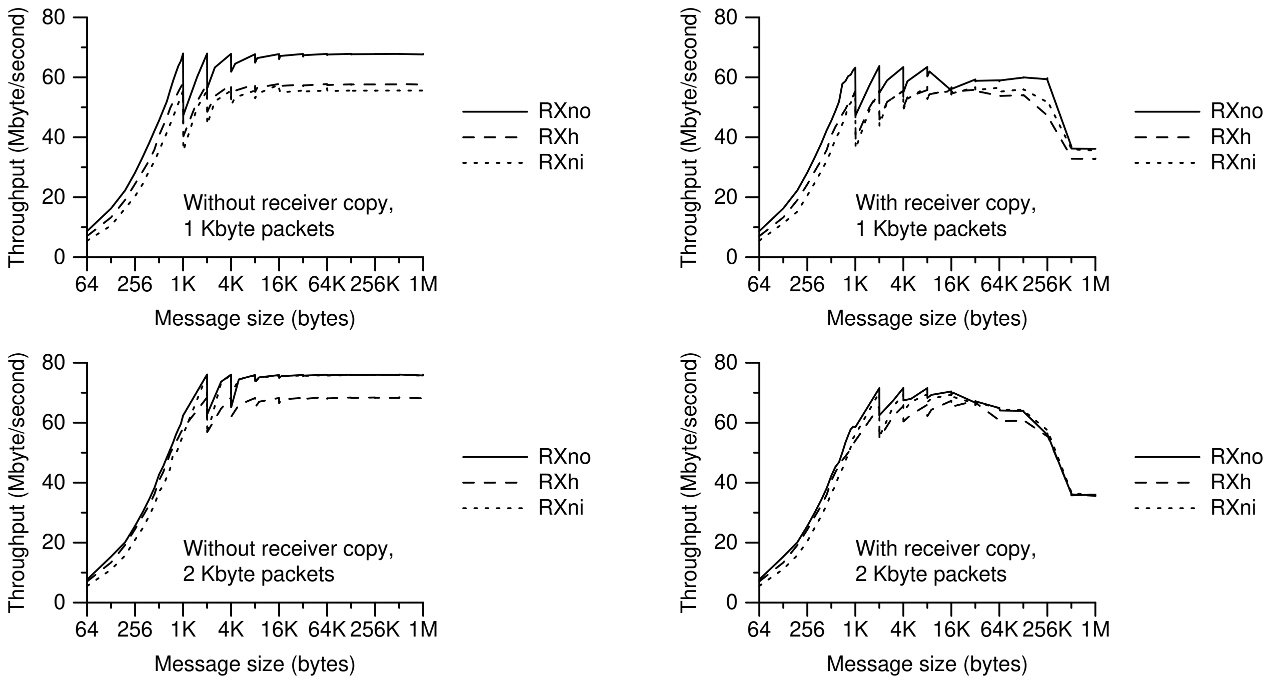

Figure 2: Unicast throughput.

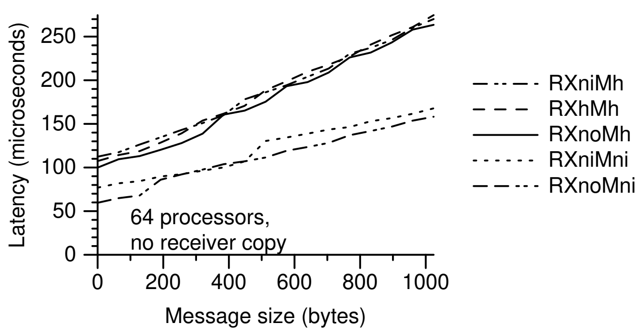

Figure 3: Broadcast latency on 64 processors (1 Kbyte packets, 8-packet send window).

RX_hM_h this work is performed more efficiently on the host. RX_hM_h's extra host-to-NI data transfer does not increase the throughput bottleneck, because it is performed in parallel with the two data transfers from the NI to its children. While this extra data transfer is invisible in this benchmark, it does consume processor cycles that an application might need (see Section 6).

Large messages do not fit in the L2 cache and make the copying of incoming packets more expensive. Added to RX_hM_h's higher receive overhead, these increased copying costs turn the receiving host into the bottleneck: RX_hM_h does not manage to copy a packet completely before the next packet arrives. This causes the dip in the RX_hM_h curve.

With a packet size of 2 Kbyte, all implementations attain higher throughput (up to 31 Mbyte/s).

During a spanning-tree broadcast, packets travel through the tree in parallel and may contend for the same physical network links. The throughput results are therefore sensitive to the way processes are mapped to processors. The results in Figure 4 were obtained with a mapping that optimizes processor 0's broadcast throughput. With other mappings, the throughput can be significantly lower.

5. PARALLEL-PROGRAMMING SYSTEMS

Each of our applications runs on one of four parallel-programming systems (see Figure 5). These PPSs use different programming paradigms: CRL [15] is an invalidation-based distributed shared-memory (DSM) system, MPI is a message-passing system, Orca [3] is an update-based DSM system, and Multigame [23] is a distributed-search system. MPI, Orca, and Multigame use Panda, a communication library designed for the development of PPSs. Below we describe Panda and each PPS.

5.1 Panda

Panda provides a message datatype, Remote Procedure Call (RPC), message passing, broadcast, and totally-ordered broadcast. To send a totally-ordered broadcast message, the sender fetches a global sequence number from a central sequencer node using LCI's fetch-and-add. Next, the sender broadcasts the message with the sequence number.

Panda copies data only when necessary. Senders supply I/O vectors; Panda copies the referenced client data directly into NI send buffers. At the receiving side, Panda queues incoming LCI packets until they can be copied to a destination specified by the receiver.

Panda can be configured with or without threads. MPI and Multigame use Panda-*S*ingle*T*hreaded (Panda-ST). Panda-ST performs no locking and is faster than Panda-*M*ulti*T*hreaded (Panda-MT). Also, Panda-ST disables network interrupts and relies on polling by its clients to receive messages. Panda-MT polls automatically when all threads are idle; otherwise messages are received by means of interrupts.

5.2 MPI

MPI is an elaborate message-passing standard. We ported MPICH, a widely used MPI implementation [13], to Panda-ST.

An MPI implementation usually generates one message for each

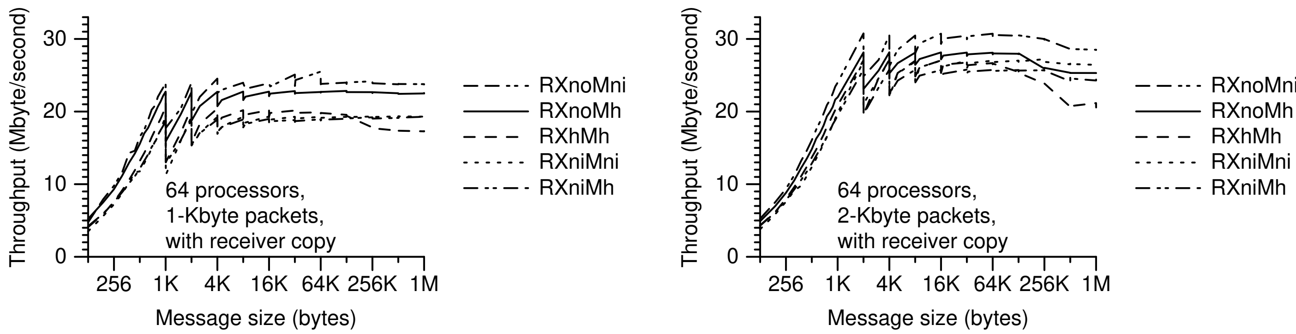

Figure 4: Broadcast throughput on 64 processors.

send statement in an MPI program. Collective operations such as broadcasts and reductions, however, involve groups of processes and generate more complex communication patterns. All our MPI applications use such operations and all these operations use a broadcast in their implementation. MPICH provides an inefficient broadcast built on top of message-based unicast primitives. We replaced this broadcast with Panda's unordered broadcast which, in turn, uses LCI's broadcast. In contrast with the MPICH broadcast, all LCI broadcast implementations forward packets instead of messages and avoid repeated host-to-NI copying (see also Section 3.4).

5.3 Orca

Orca is an object-based DSM system. To communicate, processes invoke user-defined operations on shared objects [3]. Orca requires multithreading and therefore uses Panda-MT. Orca's runtime system stores each shared object in a single processor's memory or replicates it in the memory of all processors that have a reference to the object. When a process performs an update operation on a shared object, the runtime system ships the operation's parameters to all processors that store the object; each receiving processor performs the operation. For nonreplicated objects Orca uses Panda RPC to ship the operation; for replicated objects Orca uses Panda's totally-ordered broadcast.

5.4 Multigame

Multigame (MG) is a parallel game-playing system. Given the rules of a board game and a board evaluation function, it searches for good moves. To avoid re-searching, positions are cached in a distributed hash table. To access a remote table entry, a process sends its 32-byte current-job descriptor to the entry's owner and starts working on another job [23]. The owner looks up the entry and continues to work on the job.

Table-access messages are small, one-way messages to random destinations. A process that has enough work need not wait for messages; latency in the LogP sense (L) is therefore relatively unimportant. Instead, performance is dominated by send and receive overhead. To reduce the impact of these overheads, Multigame aggregates messages before transmitting them.

5.5 CRL

CRL [15] is a software DSM system, which we ported to LCI. Processes can share memory regions of a user-defined size. Processes enclose their accesses to shared regions by calls to the CRL runtime system, which implements coherent region caching. Most communication results from read and write misses, which generate a small request to a region's *home node*, zero or more invalidations from the home node to other sharers, and a reply from the home node to the node that missed. Whether a reply carries region data

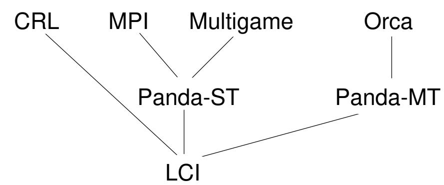

Figure 5: Dependencies between software systems.

depends on the type of miss. This single-threaded CRL implementation blocks the application during an outstanding region miss.

5.6 PPS Performance

Table 5 summarizes the minimum latency and the maximum throughput of PPS-specific operations. The table also shows the cost of similar communication patterns when executed at lower software levels. These numbers were obtained using benchmarks that send messages of the same size as the PPS-level benchmark, but do not perform PPS-specific actions such as updating coherence-protocol state. The results for different levels are separated by slashes; performance differences between levels are caused by layer-specific overheads. The percentage between brackets is the performance loss between the LCI and the PPS level. The LCI numbers were measured on $RX_{no}M_{ni}$ with a packet size of 1 Kbyte. Since CRL and Multigame use broadcasting only for operations that are not performance-critical in our applications (e.g., statistics gathering), there are no natural broadcast benchmarks for these PPSs.

For CRL, we show the cost of a clean write miss (a small control message to the home node followed by a reply with data). For MPI, we show the cost of a one-way message and the cost of a broadcast. For Orca, we show the cost of an operation on a remote, nonreplicated object (which results in an RPC) and the cost of an operation on a replicated object (which results in an F&A operation followed by a broadcast). No results are shown for the Multigame level, because a Multigame rule set does not correspond to a simple communication pattern.

A key observation is that PPS-level latencies are up to 124% larger than LCI-level latencies. This is due to (de)multiplexing, fragmentation and reassembly, locking, and procedure calls in Panda and the PPSs. Since these relatively high overheads are independent of the underlying LCI implementation, they reduce the relative differences in latency at the PPS level. The impact of PPSs on throughput is much smaller: the PPSs decrease LCI-level throughput (with receiver-side copying) by no more than 10%, because they all carefully avoid unnecessary copying.

Similar results are obtained with 2-Kbyte packets (not shown).

PPS	Latency (μs)				Throughput (Mbyte/s)			
	Unicast		Broadcast		Unicast		Broadcast	
LCI/CRL	21/26	(+25%)	–/–		59/54	(-9%)	–/–	
LCI/Panda-ST/MPI	10/15/23	(+119%)	61/69/77	(+27%)	64/62/62	(-2%)	26/26/25	(-1%)
LCI/Panda-ST/Multigame	10/15/–		–/–/–		64/62/–		–/–/–	
LCI/Panda-MT/Orca	21/32/47	(+124%)	61/93/113	(+86%)	55/53/51	(-8%)	26/26/25	(-4%)

Table 5: PPS performance summary. All measurements were performed on $RX_{no}M_{ni}$ with a packet size of 1 Kbyte. All broadcast measurements were taken on 64 processors.

Application	PPS	Problem size		1 Kbyte packets		2 Kbyte packets	
			T_1	S_{64}	E_{64}	S_{64}	E_{64}
Awari	Orca	13 stones	459.08	32.1	0.50	31.6	0.49
Barnes-Hut	CRL	16,384 bodies	123.25	23.3	0.36	22.9	0.36
Linear equation solver (LEQ)	Orca	1000 equations	640.25	30.2	0.47	31.7	0.50
Puzzle-4	MG	15-puzzle, $\leq$ 4 jobs/message	261.00	40.2	0.63	40.4	0.63
Puzzle-64	MG	15-puzzle, $\leq$ 64 jobs/message	261.00	48.4	0.76	48.7	0.76
QR factorization	MPI	1024×1024 doubles	53.47	44.3	0.69	46.9	0.73
Radix sort	CRL	3,000,000 ints, radix 128	4.40	13.1	0.21	14.0	0.22
Successive over-relaxation (SOR)	MPI	1536×1536 doubles	28.01	45.8	0.72	46.0	0.72

Table 6: Application characteristics and timings on $RX_{no}M_{ni}$, with two packet sizes. T_1 is the execution time (in seconds) on one processor; S_{64} is the speedup on 64 processors; $E_{64} = S_{64}/64$ is the parallel efficiency on 64 processors.

Latencies do not change, so the PPS impact on latency remains the same. The LCI-level throughputs increase (see Section 4), so that the relative PPS impact on throughput decreases.

6. APPLICATION PERFORMANCE

To assess the impact of different protocol decompositions, we performed application measurements on 64 processors, using the configurations shown in Table 3. Our applications range from fine-grain to medium-grain and are either unicast-dominated or multicast-dominated. Table 6 lists, for each application, its PPS, input parameters, sequential execution time, speedup, and parallel efficiency. Sequential execution time and parallel efficiency were measured using $RX_{no}M_{ni}$. Although we use small input sets, most applications achieve an efficiency of at least 50%.

Figure 6 gives per-processor data and packet rates for each application, broken down according to packet type. This figure shows the diversity of the applications' communication patterns. Data and packet rates refer to *incoming* traffic, so a broadcast is counted as many times as the number of destinations (63). These rates were measured using $RX_{no}M_{ni}$; the rates on other implementations show similar differences between applications.

Figure 7 shows application performance on $RX_{no}M_h$, $RX_{ni}M_{ni}$, $RX_{ni}M_h$, and RX_hM_h relative to the performance on $RX_{no}M_{ni}$. $RX_{no}M_{ni}$ always performs best. Below, we discuss the performance results in detail.

6.1 Awari

Awari creates an endgame database for Awari, a two-player board game. The program starts with the game's endpositions and then makes *unmoves* up to 13 levels deep. When a processor updates a board's game-theoretical value it must recursively update the board's ancestors in the unmove graph. Since the boards are randomly distributed across all processors, a single update may result in several remote update operations. To reduce communication overhead, Awari aggregates remote updates. Awari's performance is determined by the RPCs that are used to transfer the accumulated updates.

Since Awari generates few broadcast messages, Figure 7 shows no large performance differences between interface-level and host-level multicast forwarding. Retransmission support, however, slows down $RX_{ni}M_h$ by up to 13%. Although Awari's data rate appears low (see Figure 6), communication takes place in specific program phases. In these phases, communication is bursty and most messages are small despite message combining. Performance is therefore dominated by occupancy rather than roundtrip latency. The LogP parameters in Table 4 show that the RX_{ni} reliability scheme yields the highest gap ($g = 9.9\ \mu$s), followed by RX_h ($g = 8.3\ \mu$s) and then RX_{no} ($g = 6.6\ \mu$s). This ranking is reflected in the application's performance.

6.2 Barnes-Hut

Barnes simulates a galaxy using the Barnes-Hut N-body algorithm. All simulated bodies are stored in a shared oct-tree. Tree nodes and bodies are represented as small CRL regions (88–108 bytes). Each processor owns a subset of the bodies. Most communication is takes place during the force computation phase. In this phase, each processor traverses the shared tree to compute for each of its bodies the interaction with other bodies or subtrees. Due to its small regions, Barnes has a low data rate (see Figure 6). The packet rate, however, is high ($\approx$ 10,000 packets/s/processor).

Even though Barnes runs on a different PPS, its performance profile in Figure 7 is similar to that of Awari. Since CRL makes little use of LCI's multicast, there is no large difference between interface-level and host-level forwarding. Again, retransmission support leads to decreased performance and again this effect is strongest for RX_{ni} which suffers most from (NI) occupancy under high loads.

6.3 Linear Equation Solver (LEQ)

LEQ is an iterative linear equation solver. Each iteration refines a candidate solution vector x_i. To produce its part of x_{i+1}, each processor needs access to all of x_i. At the end of each iteration all processors therefore first broadcast their 128-byte partial solution vectors. Next, they synchronize to decide on convergence.

In LEQ, RX_hM_h's host-level F&A implementation is partly responsible for RX_hM_h's reduced performance. Host-level F&A

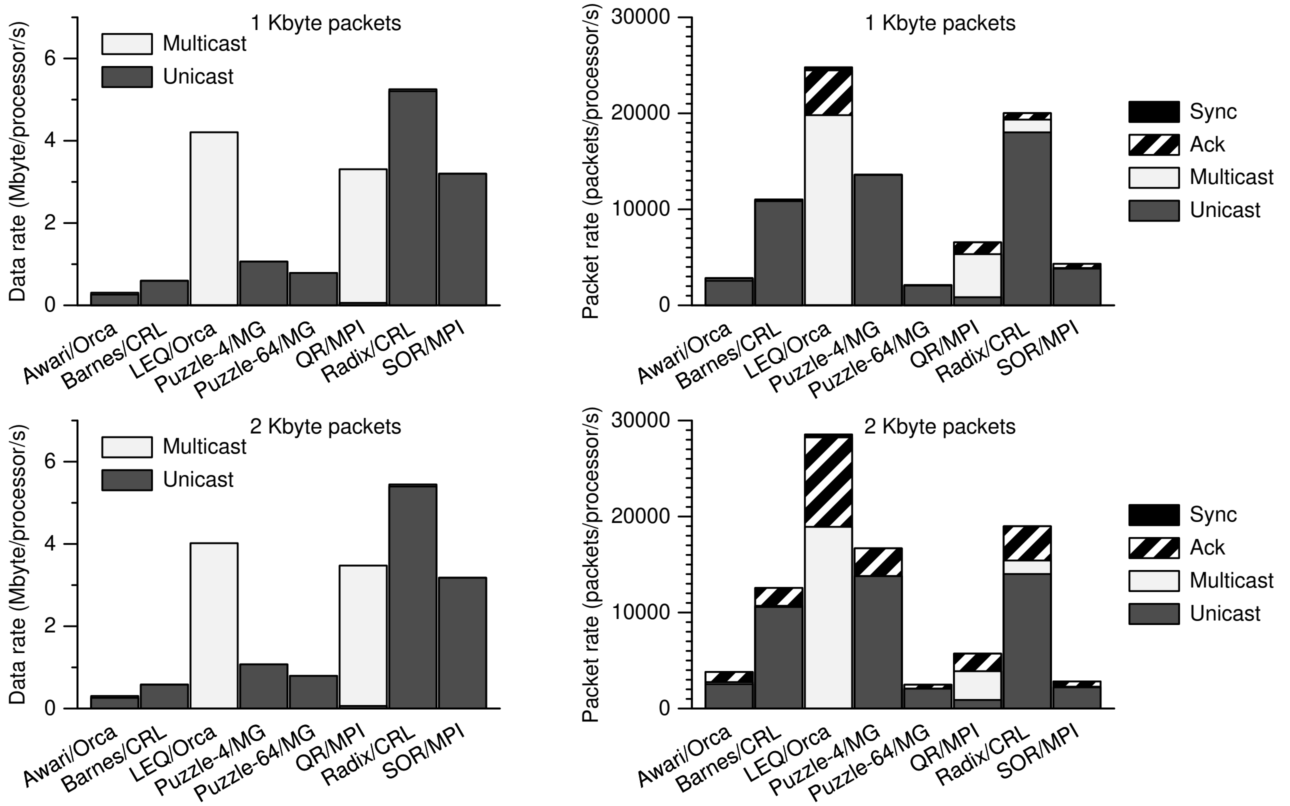

Figure 6: Data and packet rates of $RX_{no}M_{ni}$ on 64 nodes, with two packet sizes.

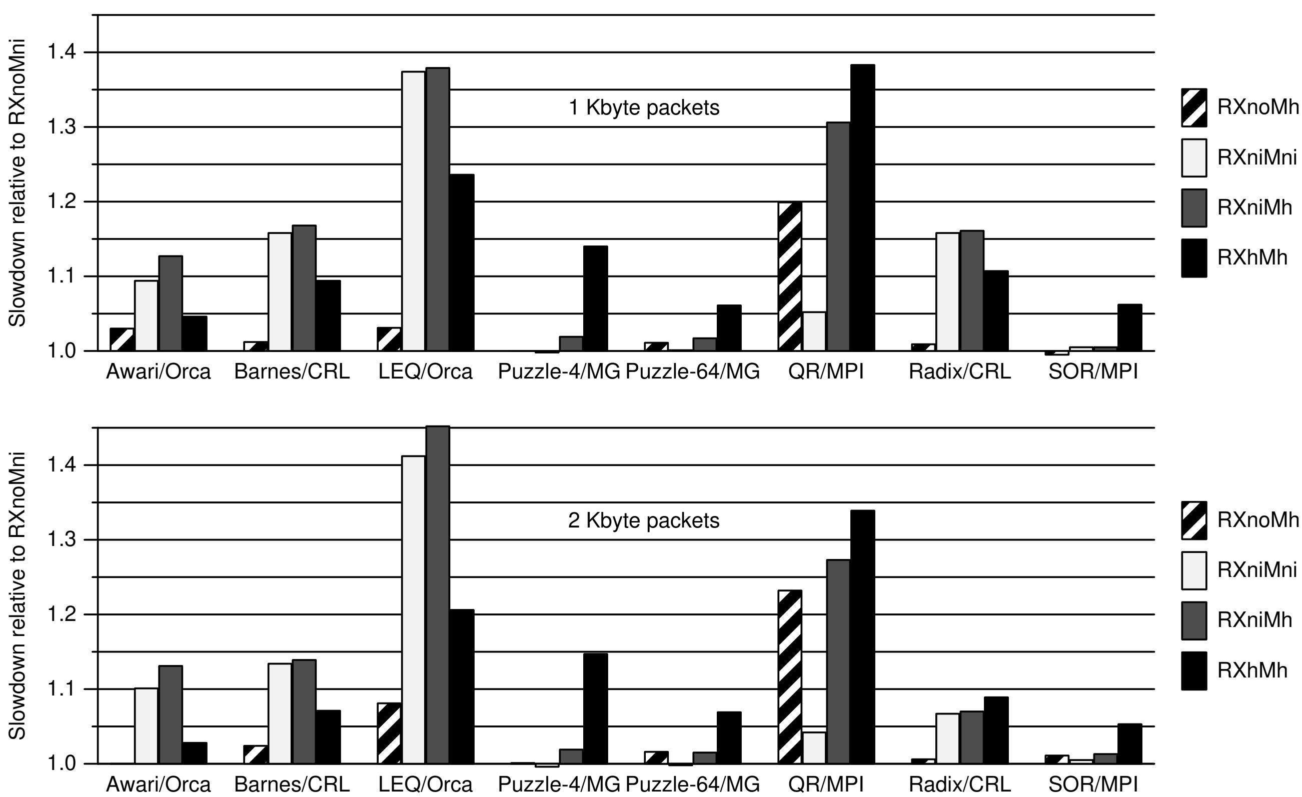

Figure 7: Application execution times, normalized with respect to $RX_{no}M_{ni}$.

processing increases the response time (see Table 4) and the occupancy of the processor that stores the F&A variable (processor 0).

Although LEQ is dominated by totally-ordered broadcast traffic, the performance differences between host-level forwarding and NI-level forwarding are relatively small. Instead, LEQ's performance profile is similar to that of the unicast applications Awari and Barnes: NI-level retransmission yields the largest performance decrease, followed by host-level retransmission. In retrospect, this is not very surprising. Since all processes simultaneously broadcast small messages, host and NI occupancy determine LEQ's performance.

6.4 Puzzle-4 and Puzzle-64

Puzzle performs a parallel IDA* search to solve the 15-puzzle, a single-player sliding-tile puzzle. Puzzle-4 aggregates at most 4 jobs before pushing them to another processor; Puzzle-64 accumulates up to 64 jobs. Both programs solve the same problem, but Puzzle-4 sends many more messages than Puzzle-64.

Since all communication is one-way and since processes do not wait for incoming messages, send and receive overhead are more important than NI-level latency and occupancy. Since RX_hM_h has the highest send and receive overheads (see Table 4), it performs worst. In Puzzle-4, the difference between RX_hM_h and other implementations is larger than in Puzzle-64. Since Puzzle-4 uses more messages than Puzzle-64 to transfer the same amount of data, it incurs the higher send and receive overheads more often.

6.5 QR Factorization

QR is a parallel implementation of QR matrix factorization. In each iteration, one column, the *Householder vector* H, is broadcast to all processors, which update their columns using H. The current upper row and H are then deleted from the data set so that the size of H decreases by 1 in each iteration. The vector with maximum norm becomes the Householder vector for the next iteration. This is decided with a reduce-to-all collective operation to which each processor contributes two integers and two doubles.

The broadcasts of column H dominate QR's performance. Figure 7 shows that the implementations based on host-level forwarding are up to 38% slower than those based on interface-level forwarding. Host-level forwarding reduces the time available for executing application code and increases broadcast latency. Broadcast latency is important, because at the start of each iteration, each receiving processor must wait for an incoming broadcast. There is no opportunity to pipeline broadcasts and hide this latency, because the reduce-to-all operation synchronizes all processors in each iteration. Also, due to pivoting, the identity of the broadcasting processor changes in almost every iteration.

6.6 Radix Sort

Radix sorts an array of random integers using a parallel radix sort algorithm. Each processor owns a contiguous part of the array. Each part is subdivided into 1-Kbyte CRL regions, which act as software cache lines. Communication is dominated by the permutation phase, in which each processor moves integers in its own array partition to other partitions. This phase is very communication-intensive and leads to Radix's high data rate. After the permutation phase, each processor reads the new values in its partition and starts the next sorting iteration.

Although Radix sends larger messages than Awari, Barnes, and LEQ, it has a similar performance profile as these applications. Radix's messages are still relatively small (at most 1 Kbyte of region data), so that Radix remains sensitive to the small-message bottleneck. With a packet size of 1 Kbyte, Radix's data messages require two LCI packets, a full packet and a nearly empty packet. With a packet size of 2 Kbyte, all messages fit in a single LCI packet. This is particularly beneficial to the implementations that use NI-level retransmission, because most processing for data packets can now be overlapped with DMA transfers.

6.7 Successive Overrelaxation (SOR)

SOR solves discretized Laplace equations. The program uses red-black iteration to update a 1536×1536 matrix. Each processor owns an equal number of contiguous matrix rows. In each iteration, processors exchange border rows (12 Kbyte) and perform a single-element reduction to decide on convergence.

RX_hM_h suffers from sliding-window stalls. Since all processes send a row to their neighbor at approximately the same time, no receiver transmits its half-window acknowledgements fast enough to prevent window stalls. The acknowledgement is not sent until the receiver itself stalls and needs to poll. In the other implementations, the sliding window is on the NI so that the host can copy packets to the NI even if the send window has closed. Also, if an NI's send window to one NI closes, that NI can still send packets to other NIs.

6.8 Discussion

The application measurements show that one LCI implementation, $RX_{no}M_{ni}$, consistently performs best. $RX_{no}M_{ni}$ assumes reliable network hardware and uses an NI-supported multicast. On other implementations, the broadcast applications are up to 45% slower and the unicast applications are up to 17% slower. Depending on the PPS and the application, $RX_{no}M_{ni}$ achieves better performance due to reduced NI occupancy, reduced send and receive overhead, or NI-level forwarding. Other implementations get some of these benefits, but not all.

Awari, Barnes, and Radix send mainly *roundtrip* messages. Nevertheless, the relative performance of the LCI implementations for these applications is predicted best by the implementations' small-message bottleneck (g) and not by their contention-free end-to-end latency. This bottleneck is largest for the implementations based on NI-level retransmission. The bursty and roundtrip nature of these applications makes them sensitive to occupancy-induced increases in end-to-end latency. Puzzle, in contrast, sends one-way messages and operates asynchronously. Puzzle is therefore most sensitive to send and receive overhead and benefits from moving the reliability protocol to the NI.

The broadcast applications also show how NI support can both increase and decrease performance. Despite the slow NI processor, NI-level multicast forwarding improves the performance of QR, because it reduces multicast latency and increases the time available to the application. In LEQ, however, the NI-level retransmission protocol turns the NI into a bottleneck during LEQ's bursty broadcast phases.

Increasing the packet size increases microbenchmark-level throughput, but does not significantly change application performance. The speedup of applications on $RX_{no}M_{ni}$ is hardly affected (see Table 6) when the packet size is increased. The performance of the other LCI implementations improves, but not dramatically (see Figure 7).

To understand this, consider Figure 8. This figure compares, for each application, the number of packets communicated when the application is run on $RX_{no}M_{ni}$ with a packet size of 1 Kbyte with the number of packets communicated when the application is run on $RX_{no}M_{ni}$ with a packet size of 2 Kbyte. The figure shows that most applications send as many data packets with a packet size of 2 Kbyte as they do with a packet size of 1 Kbyte. These applications send only small packets and therefore do not bene-

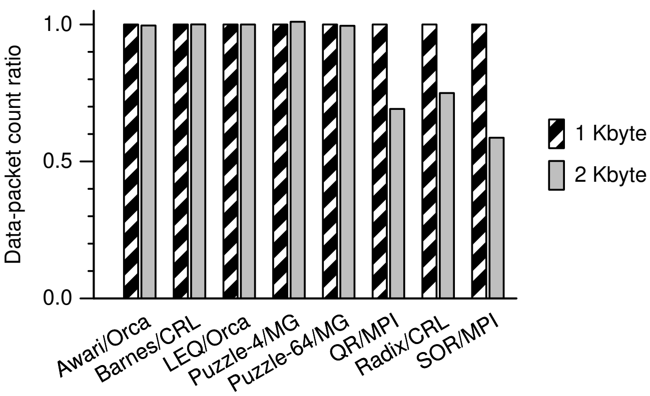

Figure 8: Data-packet count ratios for 1-Kbyte and 2-Kbyte configurations of $RX_{no}M_{ni}$.

fit from an increased packet size. On $RX_{no}M_{ni}$ and $RX_{no}M_h$, these applications might even be disadvantaged by the smaller send window that results from the increased packet size. Since we use half-window acknowledgements, acknowledgements will be sent sooner when the window shrinks and piggybacking becomes less effective. Indeed, with $RX_{no}M_{ni}$, several applications send many more acknowledgements with a packet size of 2 Kbyte than they do with a packet size of 1 Kbyte (see Figure 6). This effect occurs only with $RX_{no}M_{ni}$ and $RX_{no}M_h$ and not with the retransmitting implementations, which use the same window size for all configurations.

Only QR, Radix, and SOR send fewer packets when the packet size is increased, but this has no large impact on application performance, except for Radix on $RX_{ni}M_{ni}$ and $RX_{ni}M_h$. In QR and SOR, performance is dominated by broadcast latency and sliding-window stalls, not by per-packet overhead.

7. RELATED WORK

Karamcheti and Chien [16] studied the division of protocol tasks between network hardware and host software for CM-5 Active Messages, a messaging layer similar to LCI. They argue for higher-level services (ordering, reliability, flow control) in the network hardware to reduce costs in the software messaging layer. Our work also considers multicast *and* it considers the impact of different work decompositions on PPSs and applications.

Krishnamurthy et al. studied the role of programmable NIs in different implementations of a single PPS, Split-C [18]. Their work focusses on NI support for remote memory access and discusses neither reliability nor multicast; our work studies the implementation of a general-purpose message-passing layer.

LCI's reliability and multicast implementations make similar assumptions as existing and proposed protocols. Fast Messages [22] and PM [24] assume that the hardware never drops or corrupts packets. Active Messages II combines an NI-level reliability protocol with a host-level sliding window protocol for reliability and flow control [9]. Several papers describe NI-supported multicast protocols [5, 14, 17, 25]. We are the first to compare efficient NI-level and host-level multicasts *and* their impact on application performance.

Araki et al. used LogP [10] measurements to several user-level communication systems [1] with different reliability strategies. They compare systems with different programming interfaces (e.g., memory-mapped communication and message passing) and do not consider multicast. We compare implementations of *one* interface and also consider the application-level impact of different reliability and multicast designs.

Martin et al. studied the sensitivity of Split-C applications to LogGP parameter values [20]. They varied individual LogGP parameters using delay loops in a single communication system (Active Messages). We look at a much smaller set of parameter values, but we know that each set corresponds naturally to a particular protocol decomposition. We can therefore correlate parameter values and decompositions. We also consider a larger range of PPSs and show that some are more sensitive to particular parameters than others.

Bilas et al. identify bottlenecks in DSM systems [7]. Their simulation study revolves around the same layers as used in this paper: low-level communication software and hardware, PPS, and applications. Bilas et al. use memory-mapped communication and analyze the performance of page-based and fine-grained DSM systems. Our work uses packet-based communication and PPSs that implement message passing or object sharing.

A large fraction of this work is based on the first author's Ph.D. thesis [4]. Details of the LCI and PPS implementations can be found in this thesis.

8. CONCLUSIONS

We have systematically studied the performance impact of different divisions of communication tasks between host software, NI firmware, and network hardware.

For our applications, an implementation that assumes reliable network hardware and that performs NI-level multicast forwarding always performs best. With other decompositions, applications run up to 45% slower. In most cases, slowdowns are due to the cost of retransmission support.

Some have argued that protocol tasks should be performed on the host, because the NI processor is relatively slow [22]. Our work shows that this is not always the case. For all our implementations, NI-level multicast forwarding yields better application performance than host-level forwarding. NI-level retransmission, on the other hand, increases NI occupancy and decreases performance for PPSs and applications that send many small messages in bursts and that need to wait for such messages. PPSs and applications that can operate asynchronously, however, attain *better* performance with NI-level retransmission, because it reduces send and receive overhead on the host. In addition, our host-level implementation (RX_hM_h) is fairly agressive, because it uses NI memory for retransmission and does not inject multicast packets multiple times. A more conservative implementation would increase the advantage of NI-level protocol processing.

Our work also sheds light on the role of parallel-programming systems. PPSs such as Multigame, Orca, and CRL each generate a small number of characteristic communication patterns. An interesting result is that some protocol decompositions work better for some patterns than other decompositions. In fact, even though PPSs increase low-level latencies by up to 124%, performance differences due to different decompositions remain clearly visible at the application level.

9. ACKNOWLEDGEMENTS

We would like to thank Rimon Barr, Rob van Nieuwpoort, Werner Vogels, and the anonymous reviewers for their feedback on this paper.

10. REFERENCES

[1] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-Space Communication: A Quantitative Study. In *Supercomputing'98*, Orlando, FL, Nov. 1998.

[2] M. Aron and P. Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. In *Proc. of the 17th Symp. on Operating Systems Principles*, pp. 232–246, Kiawah Island Resort, SC, Dec. 1999.

[3] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Trans. on Computer Systems*, 16(1):1–40, Feb. 1998.

[4] R. Bhoedjang. *Communication Architectures for Parallel-Programming Systems*. PhD thesis, Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, June 2000.

[5] R. Bhoedjang, T. Rühl, and H. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Proc. of the Int. Conf. on Parallel Processing*, pp. 381–390, Minneapolis, MN, Aug. 1998.

[6] R. Bhoedjang, T. Rühl, and H. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, Nov. 1998.

[7] A. Bilas, D. Jiang, Y. Zhou, and J. Singh. Limits to the Performance of Software Shared Memory: A Layered Approach. In *Proc. of the 5th Int. Symp. on High-Performance Computer Architecture*, pp. 193–202, Orlando, FL, Jan. 1999.

[8] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[9] B. Chun, A. Mainwaring, and D. Culler. Virtual Network Transport Protocols for Myrinet. In *Hot Interconnects'97*, Stanford, CA, Apr. 1997.

[10] D. Culler, L. Liu, R. Martin, and C. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[11] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects'97*, Stanford, CA, Apr. 1997.

[12] M. Gerla, P. Palnati, and S. Walton. Multicasting Protocols for High-Speed, Wormhole-Routing Local Area Networks. In *Proc. of the 1996 Conf. on Communications Architectures, Protocols, and Applications (SIGCOMM)*, pp. 184–193, Stanford University, CA, Aug. 1996.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[14] Y. Huang and P. McKinley. Efficient Collective Operations with ATM Network Interface Support. In *Proc. of the Int. Conf. on Parallel Processing*, pp. 34–43, Bloomingdale, IL, Aug. 1996.

[15] K. Johnson, M. Kaashoek, and D. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the 15th Symp. on Operating Systems Principles*, pp. 213–226, Copper Mountain, CO, Dec. 1995.

[16] V. Karamcheti and A. Chien. Software Overhead in Messaging Layers: Where Does the Time Go? In *Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 51–60, San Jose, CA, Oct. 1994.

[17] R. Kesavan and D. Panda. Optimal Multicast with Packetization and Network Interface Support. In *Proc. of the Int. Conf. on Parallel Processing*, pp. 370–377, Bloomingdale, IL, Aug. 1997.

[18] A. Krishnamurthy, K. Schauser, C. Scheiman, R. Wang, D. Culler, and K. Yelick. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. In *Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 37–48, Cambridge, MA, Oct. 1996.

[19] O. Maquelin, G. Gao, H. Hum, K. Theobald, and X. Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proc. of the 23rd Int. Symp. on Computer Architecture*, pp. 179–188, Philadelphia, PA, May 1996.

[20] R. Martin, A. Vahdat, D. Culler, and T. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proc. of the 24th Int. Symp. on Computer Architecture*, pp. 85–97, Denver, CO, June 1997.

[21] D. Mosberger and L. Peterson. Careful Protocols or How to Use Highly Reliable Networks. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, pp. 80–84, Napa, CA, Oct. 1993.

[22] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, Dec. 1995.

[23] J. Romein, A. Plaat, H. Bal, and J. Schaeffer. Transposition Driven Work Scheduling in Distributed Search. In *AAAI National Conference*, pp. 725–731, Orlando, FL, July 1999.

[24] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High-Performance Communication Library. In *High-Performance Computing and Networking (LNCS 1225)*, pp. 708–717, Vienna, Austria, Apr. 1997.

[25] K. Verstoep, K. Langendoen, and H. Bal. Efficient Reliable Multicast on Myrinet. In *Proc. of the Int. Conf. on Parallel Processing*, pp. 156–165, Bloomingdale, IL, Aug. 1996.

[26] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th Symp. on Operating Systems Principles*, pp. 303–316, Copper Mountain, CO, Dec. 1995.