Reducing Data and Control Transfer Overhead through Network-Interface Support

Raoul A. F. Bhoedjang
Cornell University
Ithaca, NY, USA
raoul@cs.cornell.edu

Kees Verstoep Henri E. Bal
Vrije Universiteit
Amsterdam, The Netherlands
{versto,bal}@cs.vu.nl

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

Abstract

This paper describes generic NI-level mechanisms that reduce data and control transfer overheads in user-level communication systems. These mechanisms reduce polling overhead without reducing throughput, improve multicast performance, and reduce the frequency of network and timer interrupts. They have all been implemented in a family of low-level Myrinet communication systems. We illustrate the performance impact of these mechanisms using microbenchmarks and parallel-application measurements.

1. Introduction

This paper describes and evaluates mechanisms that reduce the data and control transfer overheads associated with three important components of user-level communication systems: polling, interrupts, and multicast. Our implementations of these mechanisms use Myrinet's [7] programmable network interface (NI) to perform simple, but crucial tasks.

User-level communication systems allow processes to detect and process incoming messages without kernel involvement by means of polling. A poll usually involves reading one or two words and is therefore much less expensive than an interrupt. Nevertheless, a naive implementation can reduce communication performance by introducing unnecessary I/O bus transfers, which can increase the cost of a poll. Section 3 compares different polling mechanism and describes an optimization called *packet shifting* that yields a fast poll and puts no unnecessary load on the I/O bus.

Many user-level communication systems neglect multicast [6] or provide it as an afterthought by layering it on top of message-based point-to-point primitives. Several systems have demonstrated the potential performance advantage of NI-level multicast support [5, 16]. Section 4 compares different multicast strategies and shows that NI-level multicast support reduces multicast latency, increases multicast throughput, and improves application performance.

Communication systems and their clients rely on interrupts for the asynchronous notification of events such as packet arrival and timer expiration. Interrupts, however, are expensive. Section 5 shows how a host and its NI can cooperatively reduce the frequency of network and timer interrupts. In the case of timer interrupts, NI support also allows us to implement timers with microsecond granularity. Fine-grain timers are becoming increasingly important [2]; we use them to implement fast acknowledgements.

Our main contribution is to show how relatively simple NI mechanisms can reduce data and control transfer overhead in a communication system. All mechanisms have been implemented in a family of communication systems that implement the same *L*ow-level *C*ommunication *I*nterface (LCI). They have played an important role in LCI's demonstrated capacity to support a variety of parallel-programming systems [4]. The mechanisms, however, are general and can be used in any message-passing system.

This paper is organized as follows. Section 2 describes LCI and its implementations. Section 3 describes an efficient implementation of polling that does not reduce throughput. Section 4 describes an efficient, NI-level multicast implementation. Section 5 describes two mechanisms that reduce the frequency of network and timer interrupts. Section 6 discusses related work and Section 7 concludes.

2. LCI

LCI is a low-level communication interface that forms the core communication layer of several parallel-programming systems. LCI provides reliable, FIFO, packet-based point-to-point and multicast communication. Packets can be received using polling or interrupts. Table 1 lists the main functions of LCI's programming interface.

The following systems have been ported to or implemented on LCI: CRL, a region-based distributed shared-

void *lci_send_alloc(void)	Allocates a send packet in NI memory and returns a pointer to its data part.
int lci_group_create(int *members, int nmemb)	Creates a multicast group and returns a group identifier. The process identifiers of all *nmemb* members are specified in *members*.
void lci_ucast_launch(int dst, void *pkt, int size)	Transmits *size* bytes of *pkt* to process *dst*.
void lci_bcast_launch(void *pkt, int size)	Broadcasts *size* bytes of *pkt*.
void lci_mcast_launch(int gid, void *pkt, int size)	Multicasts *size* bytes of *pkt* to all members of group *gid*.
void lci_poll(void)	Drains the network and calls *lci_upcall*() for each incoming packet.
void lci_intr_disable(void)	Logically disables network interrupts.
void lci_intr_enable(void)	Logically enables network interrupts.
int lci_upcall(int src, void *pkt, int size, int mcast)	LCI invokes this client-supplied function once per incoming packet. *Pkt* points to the data just received (*size* bytes) from *src*; *mcast* indicates whether or not *pkt* is a multicast packet. If *lci_upcall*() returns zero, LCI recycles *pkt* immediately; otherwise, ownership is transferred to the client.
void lci_packet_free(void *pkt)	Returns ownership of receive packet *pkt* to LCI.

Table 1. LCI's programming interface (slightly simplified)

memory (DSM) system [10]; MPICH [9], an implementation of the MPI message-passing standard; Manta, a parallel Java system [11]; and Orca, an object-based DSM system [3]. These systems have different communication requirements and generate different communication patterns [4]. Specifically, they differ in their message-size distributions, in whether they require interrupt-driven packet delivery or multicast, and in whether they communicate mainly synchronously or asynchronously.

The LCI implementations run on a 128-node Myrinet/Linux cluster. Each node contains a 200 MHz Intel Pentium Pro processor with 128 Mbyte of DRAM. A Myrinet NI is attached to the host's 32-bit, 33 MHz PCI bus. The NI has a 33 MHz LANai 4.1 RISC processor, 1 Mbyte of SRAM, and three DMA engines. The NIs are connected via a 3D grid of 8-port switches and full-duplex 1.28 Gbit/s links. Myrinet provides no hardware support for multicast. Each implementation gives at most one user process access to the Myrinet NI. NI access is unprotected.

The NI-supported mechanisms described in this paper have been implemented in three LCI implementations: LCI-ni, LCI-host, and LCI-mixed. (In previous publications [5, 6], LCI-ni is called LFC.) The term 'LCI' is sometimes used in statements that pertain to *all* implementations. A detailed comparison of the implementations is given elsewhere [4]; here we summarize their main characteristics.

LCI-ni makes aggressive use of the Myrinet hardware. LCI-ni does not implement retransmission, assuming that the hardware neither drops nor corrupts packets. To prevent NI-level buffer overflows, LCI-ni runs an NI-level flow control protocol. Moreover, LCI-ni uses Myrinet's programmable NI to provide an efficient multicast implementation [5]. LCI-host implements both retransmission and multicast forwarding on the host. LCI-mixed does not implement retransmission (like LCI-ni) and implements multicast forwarding on the host (like LCI-host).

Figure 1 illustrates LCI's control and data flow path for point-to-point packets. This path is identical in all three implementations. To send a point-to-point packet, an LCI client calls *lci_send_alloc*() to allocate a 1 Kbyte send buffer in NI memory (step 1 in Figure 1) and uses programmed I/O to copy data into this buffer (step 2). Next, the client calls *lci_ucast_launch*(), which creates a send descriptor in NI memory (step 3). When the NI finds the descriptor, it transmits the packet (step 4). The receiving NI copies the packet to pinned host memory, using DMA (step 5).

If network interrupts are enabled, the receiving NI may generate a network interrupt (step 6; see also Section 5.1). The kernel dispatches network interrupts as Unix signals to the receiver process. Each signal is processed by LCI's signal handler, which calls a polling routine to process packets queued in host memory.

If interrupts are disabled, packets in host memory are detected through polling. The polling routine, *lci_poll*(), passes each packet to a client-supplied upcall routine named *lci_upcall*() (step 7). This routine processes the packet in a client-defined way and then either releases the packet or queues it for further processing. Since host buffers are stored in pinned memory —a relatively scarce resource— clients should not queue packets unnecessarily. Queued packets must be released explicitly by calling *lci_packet_free*().

3. NI-Level Packet Shifting

Packet shifting is an NI-supported mechanism that reduces the cost of polling without reducing throughput. It is used by all LCI implementations.

3.1. Polling on Myrinet

Polling is the control-transfer mechanism of choice in most user-level communication systems, because interrupts

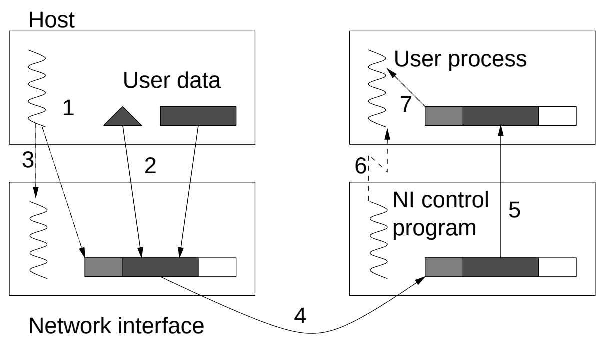


Figure 1. Data and control transfer in LCI

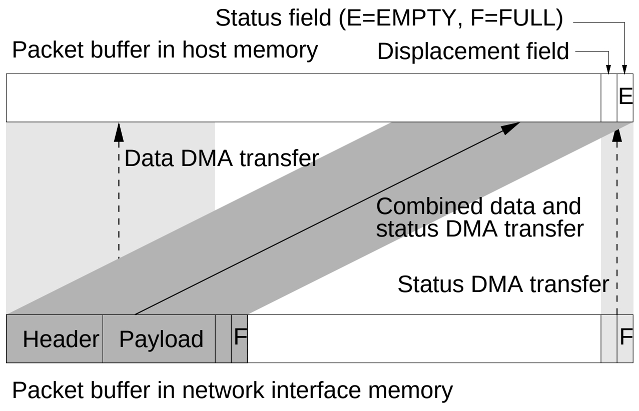
are expensive. On a 200 MHz Pentium Pro running Linux, for example, dispatching an interrupt to a kernel interrupt handler costs approximately 8 μs. Dispatching to a user-level signal handler costs an additional 17 μs. This cost exceeds LCI-ni's one-way null latency (12 μs).

A simple way to implement polling on Myrinet is to let the NI increment a variable in its memory and to let the host read the variable. This is inefficient, however, because every poll may cause an I/O bus transfer. This transfer increases the time to execute the poll.

On architectures with cache-coherent DMA, these I/O bus transfers can be avoided by letting the NI update a variable in cached host memory. Each time the NI has transferred a packet to host memory, it updates a host variable by means of a second, small DMA transfer. The host polls by reading this variable from its local memory. If polls are executed frequently, the flag will be moved into the data cache. Failed polls are therefore inexpensive and generate neither memory nor I/O traffic. When the NI writes the flag, the host will incur a cache miss and read the flag's new value from memory. On a 200 MHz Pentium Pro, the scheme just described costs 5 nanoseconds for a failed poll (i.e., a cache hit) and 74 nanoseconds for a successful poll (i.e., a cache miss). For comparison, each poll in the simple scheme (i.e., an I/O bus transfer) costs 467 nanoseconds.

3.2. Polling in LCI

LCI uses an optimized variant of the DMA-based polling approach described above. In LCI, each host receive buffer contains a status field that indicates whether the control program has copied data into the buffer. Receive buffers are organized in a queue. During a poll, LCI tests the status field of the buffer at the head of the queue. When the NI receives a packet, it must copy the packet's payload into a host receive buffer and set that buffer's status field to FULL. For efficiency, LCI folds both actions into a single DMA transfer. This is illustrated in Figure 2, which also shows the simpler implementation that uses two DMA transfers.



Status field (E=EMPTY, F=FULL)
Packet buffer in host memory
Displacement field
E
Data DMA transfer
Combined data and status DMA transfer
Status DMA transfer
Header
Payload
F
F
Packet buffer in network interface memory

Figure 2. Packet shifting

The simplest way to avoid two DMA transfers is to copy the whole NI packet buffer to host memory, but this is inefficient for packets with a small payload. To avoid transferring the empty part of an NI receive buffer, the status field must be stored right after the packet's payload. (If it were stored in front of the data, the host could believe it received a packet before all of that packet's data had arrived.) The position of the FULL word therefore depends on the size of the packet's payload. The host, however, needs to know in advance where to poll for the status field. LCI therefore transfers the packet's payload and the FULL word to the *end* of the host buffer. In addition, LCI transfers a word that holds the payload's displacement relative to the beginning of the host receive buffer. Using the displacement, the host can directly compute the start of the packet's data.

This *packet shifting* requires only a few simple calculations on the NI, works well for large and small packets, and allows the host to poll without generating I/O bus traffic.

3.3. Performance Impact

Figure 3 shows the impact of different polling strategies on small-message throughput, an important performance metric for communication systems that support parallel programming. 'Packet shifting' and 'Programmed I/O' attain the highest throughput. With 'Programmed I/O,' the host polls a counter in NI memory. This strategy achieves high throughput, but increases the latency of individual polls. The 'Full packet' strategy always uses a single DMA transfer to copy an entire 1 Kbyte NI receive buffer to the host, irrespective of the actual payload. This yields poor throughput for small packets and throughput remains worse than with packet shifting until packets are nearly full. The 'Double DMA' strategy always uses two DMA transfers, one for the data and one for the FULL word. This yields poor throughput, especially for packets that are more than half-full. The 'Static split' strategy is a combination of 'Double

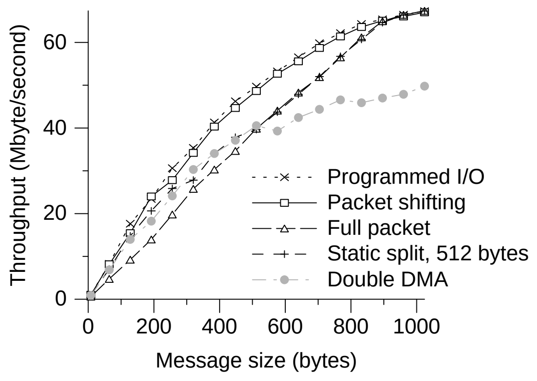


Figure 3. Comparison of polling strategies

DMA' (for packets with a payload of less than 512 bytes) and 'Full packet' (for packets with a larger payload). This strategy outperforms 'Full packet' for small payloads, but still lags behind 'Packet shifting' for larger payloads.

All variants achieve their peak throughput with 1 Kbyte messages and maintain that throughput for larger messages, with the exception of some dips caused by fragmentation.

4. NI-Level Multicast Forwarding

LCI provides a reliable, FIFO multicast primitive. Multicast is used to implement collective communication operations in message-passing systems and to update replicated data in DSM systems [3]. Since Myrinet does not support multicast in hardware, communication systems must implement multicast services in software. This is usually done by means of a spanning-tree multicast protocol. In such a protocol, the sender and receivers are organized into a multicast tree. The sender is the root of the tree and transmits each multicast packet to all its children (a subset of the receivers). These children, in turn, forward each packet to their children, and so on. Tree-based protocols allow packets to travel in parallel along the branches of the tree and therefore usually have logarithmic complexity.

4.1. Host-Level Multicast Forwarding

Most communication systems implement multicast forwarding using point-to-point message-passing primitives. Each host in the multicast tree receives an incoming multicast as a point-to-point message and then forwards this message to its children.

LCI-host and LCI-mixed use an optimized version of this scheme. First, both implementations forward individual packets rather than entire messages: forwarding begins as soon as the first packet of a multipacket message has been received. Systems that layer multicasting over message-based point-to-point primitives delay forwarding until an entire message has been received, which increases latency and reduces throughput for multipacket messages. Second, during forwarding, both implementations copy packets back to the NI only once, even when they forward to multiple children. Instead of copying the data multiple times, multiple send descriptors are created, instructing the NI to send the same packet to multiple destinations. Together, these optimizations allow LCI-host and LCI-mixed to attain fairly good multicast throughput (see Section 4.3).

4.2. NI-Level Multicast Forwarding

Even with the optimizations described above, host-level forwarding has drawbacks. First, since each reinjected packet was already available in NI memory, host-level forwarding results in an unnecessary host-to-NI data transfer at each internal tree node of the multicast tree (see Figure 4, left). Second, if one host does not poll the network in a timely manner, multicast delivery will be delayed in a whole subtree. Instead of relying on polling, the NI can raise an interrupt, but interrupts are expensive. Third, the critical sender-receiver path includes the host-NI interactions of all the nodes between the sender and the receiver. These interactions consist of copying the packet to the host, host processing, and reinjecting the packet.

To solve these problems, LCI-ni uses NI-level multicast forwarding. Figure 4 contrasts host-level (left) and NI-level (right) forwarding. NI-level forwarding does not require the host-to-NI data transfer and removes the NI-to-host transfer from the critical path: a packet can be forwarded independently of the NI-to-host copy. Here, we describe only the data transfer behavior of this NI-level multicast protocol. Deadlock avoidance and the implementation of reliability are described elsewhere [4, 5].

LCI-ni attaches a special multicast tag to multicast packets, which is recognized by all NIs. When an NI receives a multicast packet, it looks up the packet's forwarding destinations in a table stored in NI memory. Each sender has its own multicast tree; by default, all implementations use binary trees. The NI transmits the packet from its NI receive buffer to all forwarding destinations without making any internal data copies. After starting the first of these forwarding transfers, the NI also starts a DMA transfer that copies the packet to host memory. (The NI can perform two DMA transfers simultaneously.)

4.3. Performance Impact

Figure 5 shows the multicast latency and throughput on 64 processors attained by the LCI implementations. The latency benchmark measures the latency to the last receiver of a broadcast. The throughput benchmark measures the throughput observed at the sender. LCI-ni clearly attains

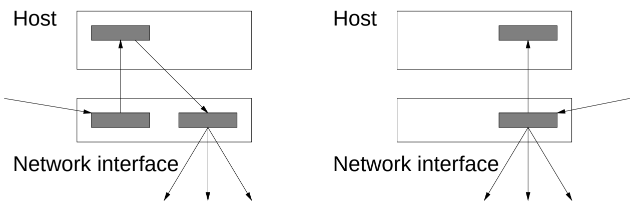
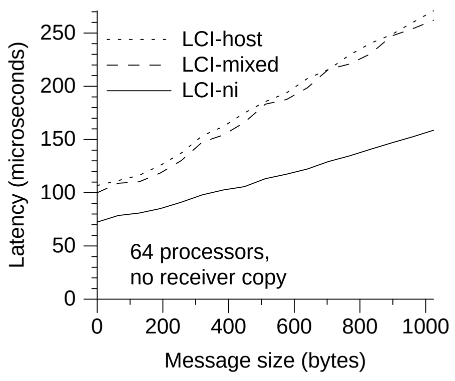


Figure 4. Multicast forwarding strategies



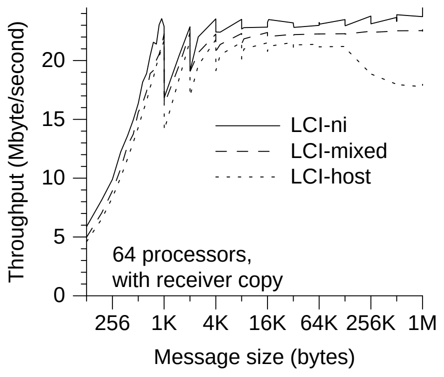


Figure 5. Multicast latency and throughput

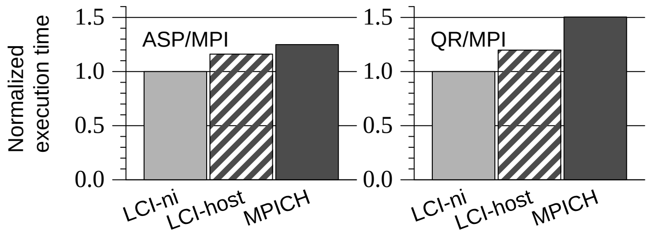


Figure 6. Comparison of multicast strategies

the lowest latency. It completely eliminates a host-to-NI copy and removes an NI-to-host copy from the critical path. LCI-ni also has the best throughput, but the difference with host-level forwarding is less dramatic than in the latency benchmark, because the host-level forwarding implementations have been heavily optimized. Moreover, this benchmark does not show the impact of spending host-processor cycles on multicast forwarding.

Figure 6 illustrates the impact of different forwarding strategies on two MPI applications, ASP and QR. Both applications were run on 64 processors and their performance is dominated by broadcast traffic. These measurements use two MPI implementations, both of which are based on MPICH [9], a widely used MPI implementation. One implementation, called 'MPICH' in Figure 6, uses MPICH's default broadcast implementation, which, in turn, uses MPI's point-to-point primitives to forward entire messages at the host level. The point-to-point primitives are implemented on top of LCI's point-to-point packet transmission facility; LCI's broadcast and multicast are not used. We modified this implementation to use binary broadcast trees instead of binomial trees, so that the implementation would use the same trees as the other two implementations.

The other MPI implementation uses LCI's broadcast instead of MPICH's default broadcast. We ran this implementation on LCI-host and LCI-ni; these variants are labeled 'LCI-host' and 'LCI-ni' in Figure 6. Both variants forward packets rather than messages, but LCI-host forwards on the host, while LCI-ni forwards on the NI.

The All-pairs Shortest Path (ASP) program finds the shortest path between all nodes in a graph. In each iteration, one processor broadcasts a 4 Kbyte matrix row. The algorithm iterates over all of this processor's rows before switching to another processor's rows. Consequently, the current sender can pipeline the broadcasts of its rows. Due to this pipelining, receivers are often still working on one iteration when broadcast packets for the next iteration arrive. MPICH and LCI-host do not process these packets until the receivers invoke a receive primitive. Consequently, the sender is stalled, because acknowledgements do not flow back in time. To improve performance, we augmented ASP with application-level polling statements (*MPI_probe*()). Figure 6 shows the improved numbers. With LCI-ni, the problem does not occur, because acknowledgements are sent by the NI, not by the host processor.

Even with application-level polling in LCI-host and MPICH, LCI-ni performs better. The remaining difference is due to the processor overhead caused by failed polls and host-level forwarding: the time spent on these activities is not available to the application.

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. At the end of each iteration, QR uses a Reduce-To-All collective operation to select the Householder vector for the next iteration. Performance is dominated by the broadcasts of column H. As with ASP, host-level forwarding increases processor overhead. In addition, the implementations based on host-level forwarding are sensitive to their higher broadcast latency. At the start of each iteration, each receiving processor must wait for an incoming broadcast. The broadcasting processor cannot pipeline multiple broadcasts, because the Reduce-to-All synchronizes all processors in each iteration. Message-based forwarding, used in MPICH's default broadcast implementation, exacerbates the latency problem and slows down QR by more than 50%.

5. NI-Level Interrupt Management

Many communication systems use and sometimes need interrupts to signal the completion of a packet's transmission, the arrival of a packet, or the expiration of a timer. Unfortunately, interrupts are expensive. Below we describe two NI-supported mechanisms that help reduce the frequency of receive and timer interrupts. In both cases, the host uses polling *when this is practical*. The NI monitors the host's polling frequency and generates interrupts when the host fails to poll within a certain period of time.

5.1. The Polling Watchdog

Many parallel-programming systems need asynchronous message delivery. Processes in DSM systems, for example, need to respond to cache-update or cache-invalidation requests that generally arrive at unpredictable times. Interrupts are a simple solution, but generating an interrupt for each incoming packet is usually not necessary. Even in a DSM system, many messages, in particular responses to requests, arrive at more or less predictable times. Ideally, these messages should be received through polling.

LCI implementations use a *polling watchdog* [12] to avoid generating interrupts unnecessarily. This mechanism *delays* interrupts in the hope that the target process will soon poll the network. Where the original polling watchdog proposal by Maquelin et al. is a hardware design, LCI uses Myrinet's programmable NI to implement a software polling watchdog. Briefly, the NI starts a watchdog timer when it receives a packet. The NI generates an interrupt only if the host does not poll before the timer expires.

LCI's polling watchdog cooperates with LCI's network-interrupt management mechanism. Recall that LCI clients may dynamically disable and enable network interrupts. For efficiency, LCI uses *optimistic interrupt protection* [14]: *lci_intr_disable*() and *lci_intr_enable*() toggle the interrupt status flag in host memory, without synchronizing with the NI, the source of network interrupts. Consequently, the NI may generate an interrupt *after* the host called *lci_intr_disable*(). The kernel will then dispatch a signal to the user process. LCI's signal handler, however, always returns without processing packets if the interrupt status flag in host memory indicates that interrupts are disabled.

LCI starts the watchdog timer each time a packet arrives and the timer is not already running on behalf of an earlier packet. If the timer expires, the NI decides whether it should generate an interrupt. This decision is based on two variables in host memory: the interrupt status flag and a count of the number of packets processed by the host. When the timer expires, the NI copies both variables to NI memory using a single, small DMA transfer. If the packet counter indicates that the host has processed all packets delivered to it by the NI, then the NI cancels the polling watchdog timer. If the interrupt status flag indicates that the host recently disabled interrupts or if the packet counter shows that the host consumed *some* of the packets delivered to it, then the NI does not generate an interrupt, but restarts the timer. Otherwise, the NI generates an interrupt and restarts the timer.

Since the NI may read a stale copy of the interrupt status flag, it may decide not to generate an interrupt after the host re-enabled interrupts. This is not a problem, because in this case the NI always restarts its watchdog timer and will generate the interrupt at a later time.

Figure 7 shows the impact of different watchdog delays on the performance of three CRL applications run on 64 processors: Barnes, FFT, and Radix. CRL is a software DSM system [10], 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. Each read or write miss generates 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. CRL normally uses network interrupts, but disables network interrupts and polls when it waits for a reply to an outstanding request.

For all applications, performance degrades when the interrupt delay grows larger than approximately 100 μs. In fact, Radix and Barnes do not terminate when interrupts are completely disabled, unless explicit polls are added to the application. Such polls, however, are not part of CRL's application programming interface. Always generating an interrupt immediately (the 'No delay' data points) also degrades performance. A delay as small as 1 μs already prevents many interrupts and improves performance. Interestingly, a large range of delays works well for these applications. Maquelin et al. [12] reported similar results.

5.2. Fine-Grain Timers

LCI-host uses timers to schedule retransmissions and fast acknowledgements. Recall that LCI stores all outgoing data in send buffers in NI memory (see Figure 1). Since LCI-host implements a retransmission-based reliability protocol, it must buffer these packets until they have been acknowledged. Since NI memory is small, however, acknowl-

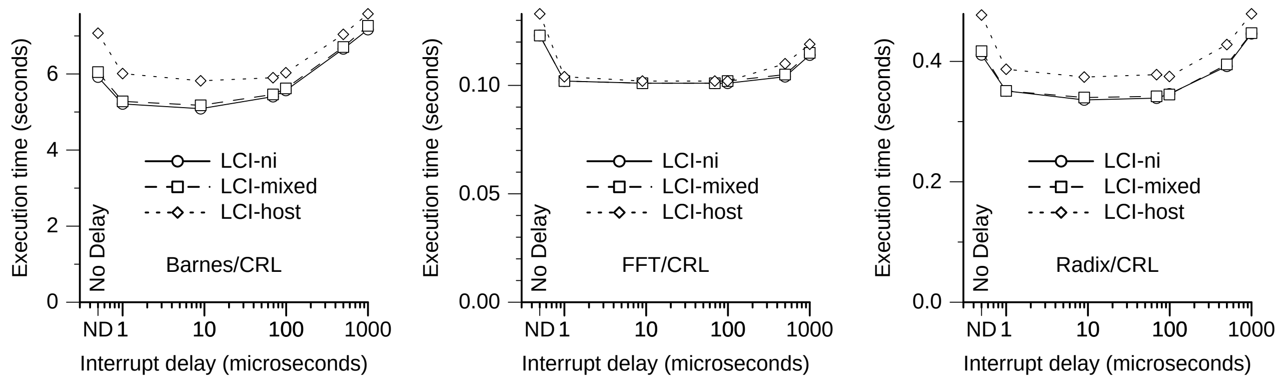


Figure 7. Impact of interrupt delay on application performance

edgements must flow back before the sender runs out of send buffer space. Receivers could explicitly acknowledge each incoming data packet, but piggybacking is more efficient. In the absence of return traffic, however, fast explicit acknowledgements are needed. Each receiver therefore maintains a per-sender timer, which is started whenever a data packet arrives from some sender. When the timer expires, an explicit acknowledgement is sent. If a packet travels back to the sender before the timeout, an acknowledgement will be piggybacked and the timer will be canceled.

The granularity of operating system timers, typically at least ten milliseconds, is too coarse for this acknowledgement timer. LCI-host therefore uses an NI-supported variant of *soft timers* [2] to implement fine-grain timers. Instead of relying only on OS timer signals, the host polls the clock (in our case, the CPU timestamp counter) . LCI-host polls the clock each time an LCI routine enters LCI's timer module and during timer interrupts. Unlike soft timers, which use OS timers as a backup mechanism against infrequent polling, LCI-host lets the NI generate periodic clock interrupts. The NI, however, generates these interrupts only when a timer is actually running or when the host has not recently polled the clock. This is checked by fetching host state (using a small DMA transfer) before generating the periodic interrupt. This state contains a count of the number of running timers and the time of the last clock poll.

Figure 8 shows the impact of different clock granularities on the performance of a Fast Fourier Transform (FFT) on CRL and on 64 processors. The horizontal axis indicates the time T between clock interrupts. If a timer has been started, the NI will generate a clock interrupt every T microseconds, unless the host frequently polls the clock. We configured LCI-host with 64 NI send buffers. FFT performs all-to-all exchanges of matrix blocks. During these exchanges, processors run out of send buffers unless they receive timely acknowledgements. Figure 8 shows clearly that performance degrades when the clock period is larger than 1 millisecond.

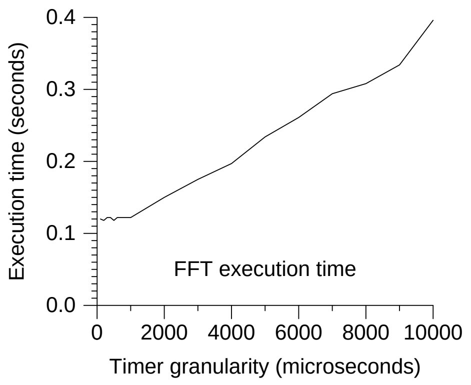


Figure 8. Impact of timer granularity

6. Related Work

Mukherjee et al. proposed *cacheable device registers* to reduce the cost of polling. A cacheable device register is a special device status register that is shared between the NI and the host [13]. Current hardware, however, does not provide these shared registers.

Several NI-level multicast protocols have been proposed [5, 8, 16]. None of the cited papers, however, demonstrates the application-level performance advantage of NI-level multicasting. Some papers compare host-level and NI-level multicasting using microbenchmarks, but they use host-level multicast schemes that are less aggressive than the scheme used by LCI-host and LCI-mixed.

Research systems such as the Alewife [1] have attempted to reduce the nominal cost of an interrupt. Even without hardware support, operating systems can dispatch interrupts more efficiently by saving less processor state (and less often) [15]. These proposals, however, have not found their way into commercial architectures and operating systems.

We do not attack the overhead of individual interrupts, but use NI support to reduce the interrupt frequency. LCI optimistically delays receive interrupts in the hope that a poll will soon occur. The polling watchdog proposed by Maquelin et al. is a hardware implementation of this idea [12]. We augmented the polling watchdog with several checks to further reduce the risk of spurious interrupts.

Our NI-supported fine-grain timers are similar to Aron and Druschel's *soft timers* [2]. They optimize kernel timers by polling the host's clock and processing timeouts during system calls, exceptions, and interrupts. This way, they amortize the state-saving overhead of these events. Their soft timers use the kernel's coarse-grain clock interrupt as a backup mechanism against low polling rates. LCI provides user-level soft timers and uses the NI as a much more precise backup source of timer interrupts.

7. Conclusions

We have described NI-level optimizations that reduce data and control transfer overheads in user-level communication systems. Packet shifting yields fast polls without compromising throughput. NI-level multicast support yields lower multicast latency, higher multicast throughput, and better application performance. A naive, message-based, host-level multicast performs up to 50% slower at the application level. A heavily optimized, packet-based, host-level multicast is up to 20% slower at the application level. Finally, we considered the role of the network interface in interrupt management. By allowing the host to poll interrupt-generating devices when this is convenient, interrupt rates can be reduced. To guarantee responsiveness, we use the network interface as a monitor that generates interrupts when the host fails to poll.

References

[1] A. Agarwal, J. Kubiatowicz, D. Kranz, B. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[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*, pages 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 Mathematics and 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 1998 Int. Conf. on Parallel Processing*, pages 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] 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.

[8] 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*, pages 184–193, Stanford University, CA, Aug. 1996.

[9] 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.

[10] 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*, pages 213–226, Copper Mountain, CO, Dec. 1995.

[11] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *7th Symp. on Principles and Practice of Parallel Programming*, pages 173–182, Atlanta, GA, May 1999.

[12] 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*, pages 179–188, Philadelphia, PA, May 1996.

[13] S. Mukherjee, B. Falsafi, M. Hill, and D. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proc. of the 23rd Int. Symp. on Computer Architecture*, pages 247–258, Philadelphia, PA, May 1996.

[14] D. Stodolsky, J. Chen, and B. Bershad. Fast Interrupt Priority Management in Operating Systems. In *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 105–110, San Diego, Sept. 1993.

[15] C. Thekkath and H. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, CA, Oct. 1994.

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