Performance of a High-Level Parallel Language on a High-Speed Network*

Henri Bal Raoul Bhoedjang Rutger Hofman Ceriel Jacobs
Koen Langendoen Tim Rühl Kees Verstoep

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Abstract

Clusters of workstations are often claimed to be a good platform for parallel processing, especially if a fast network is used to interconnect the workstations. Indeed, high performance can be obtained for low-level message passing primitives on modern networks like ATM and Myrinet. Most application programmers, however, want to use higher-level communication primitives. Unfortunately, implementing such primitives efficiently on a modern network is a difficult task, because their software overhead is relatively much higher than on a traditional, slow network (such as Ethernet).

In this paper we investigate the issues involved in implementing a high-level programming environment on a fast network. We have implemented a portable runtime system for an object-based language (Orca) on a collection of processors connected by a Myrinet network. Many performance optimizations were required in order to let application programmers benefit sufficiently from the faster network. In particular, we have optimized message handling, multicasting, buffer management, fragmentation, marshalling, and various other issues. The paper analyzes the impact of these optimizations on the performance of the basic language primitives as well as parallel applications.

Keywords: clusters, threads, communication protocols, multicast, Myrinet, Illinois Fast Messages.

1 Introduction

Due to their wide availability, networks of workstations are an attractive platform for parallel processing. A major problem, however, is their high communication cost. Workstations are typically connected by a Local Area Network (LAN) such as Ethernet, which is orders of a magnitude slower than the interconnection networks used for modern multicomputers. Potentially, this problem can be solved by using a faster, more modern LAN, such as ATM, Fast Ethernet, Myrinet [4], or SCI [11]. Unfortunately, even with fast, Gigabit/sec networks, several important performance problems remain.

First, some of the modern networks (and their software) are not designed to support parallel processing. They offer impressive bandwidths, but the communication latencies are often only marginally better than on traditional Ethernet. For parallel processing, latency is at least as important as bandwidth.

Second, to make network-based parallel computing successful, it is essential that easy-to-use programming environments are developed. Giving the programmer the ability to unreliably send a 48-byte packet from one machine to

*This research is supported in part by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

another simply won't do. A high-level programming environment, however, will most likely further increase communication latency. On a high-speed network, the software overhead introduced by a programming system may easily become the dominating factor that limits performance.

In this paper we investigate the performance issues involved in implementing a high-level parallel programming system on a high-speed network. For this purpose, we study the implementation and performance of a specific programming system (Orca) on a fast network (Myrinet). Orca provides a high-level programming model, which can be characterized as an object-based distributed shared memory. The Orca runtime system transparently migrates and replicates shared objects, and keeps replicated objects consistent. Low-level issues like buffer management and marshalling are taken care of automatically. So, Orca has a high abstraction level, making an efficient implementation a challenging task.

We have drawn two main conclusions from this case study. First, substantial performance improvements at the language level can be achieved using a fast network. The latency and throughput of our basic communication primitives is much better on Myrinet than on Ethernet. The performance lags behind that of the "raw" hardware or low-level software, because of the higher abstraction level we use. The second conclusion is that many performance problems had to be solved to obtain this improvement. We initially used an existing, highly efficient, message passing library for the low-level communication. Still, many existing as well as new optimization techniques were required at all levels of the system to efficiently implement our high-level model.

The main contributions of the paper are a discussion of which optimizations are required for high-level languages and a quantitative analysis of the impact of these optimizations on performance. We describe the performance of our system at various levels, including the low-level communication primitives, the language level primitives, and applications. Several more recent programming languages and libraries also support shared objects of some form [10, 12, 16, 19, 21, 26, 29], so we think our work is relevant to many other systems.

The outline of the paper is as follows. In Section 2 we provide more background information about the hardware and software used in the case study. In Section 3 we describe an initial implementation of the Orca system on Myrinet. We also look at the performance problems of this initial design. Next, in Section 4 we look at the various optimizations made in message handling, multicast communication, and various other issues. In Section 5 we give performance measurements of the resulting system, using benchmarks as well as parallel applications. In Section 6 we compare our work with that of others. Finally, in Section 7 we discuss the most important lessons we have learned about the implementation of high-level languages on high-performance networks.

2 Background

This section provides background information about the programming environment and network that we use. We only give information required to understand our case study; more detailed information can be found elsewhere [2, 3, 4, 31, 39].

2.1 The Orca System

Orca is a portable, high-level language for writing parallel applications that run on distributed-memory machines [2, 3]. It has been implemented on multicomputers (CM-5, Parsytec PowerXplorer) as well as on network-based distributed systems (Solaris, Amoeba). The language has been used for dozens of realistic applications.

Orca supports explicit task-parallelism in the form of sequential processes, which can be created dynamically. Processes communicate through *shared objects*, which are instances of Abstract Data Types (ADTs). The language guarantees that each operation on a shared object is executed indivisibly (as in a monitor). Each operation is always applied to only a single object. Programming with shared objects is similar to programming with shared variables, except that all operations are user-defined, indivisible ADT operations. The programming model is based on shared data, but it is implemented on a distributed-memory system. It thus can be regarded as a form of object-based distributed shared memory [31].

The implementation of the language (the compiler and the runtime system) uses two different methods to implement shared objects [2]. The compiler determines which operations are read-only and which operations may write their object. If an object is modified infrequently (i.e., most operations on it are read-only), the object will be *replicated* in the local memories of all processors that can access it. Read-only operations on the object are executed locally, without communication. An operation that writes the object is multicast to all machines, which update their local copy by executing the operation. Coherency is obtained by using *totally-ordered multicast* [17], which guarantees that all messages are received by all destinations in the same order. Alternatively, an object can be stored on a single machine. In this case, other machines can access it using remote object invocations, which are implemented using remote procedure calls (RPCs).

For portability reasons, the Orca system is structured using multiple layers. The runtime system (RTS) only deals with processes and object management, but not with implementing communication. Communication primitives are provided by a lower layer, called *Panda* [25]. Panda is a virtual machine supporting the functionality needed by the RTS (threads, totally-ordered multicast, and remote procedure calls). The Orca RTS only invokes Panda primitives, and is unaware of the underlying operating system. The Orca compiler generates ANSI C as output. Thus, to implement

the entire Orca system on a new platform, only Panda has to be ported.

On most platforms, Panda is implemented using user-space communication protocols [25]. The structure of Panda is very flexible. For example, if the hardware or operating system already provides reliability or total ordering, the Panda protocols benefit from this, thus simplifying them and making them more efficient.

2.2 Myrinet

Myrinet is a high-speed Local Area Network designed to support parallel processing on clusters of workstations [4]. Myrinet can be added to a workstation cluster to increase communication performance. It uses two types of hardware components: an interface board (host adapter) that is inserted in each workstation, and a switching network that interconnects the interface boards (see Figure 1).

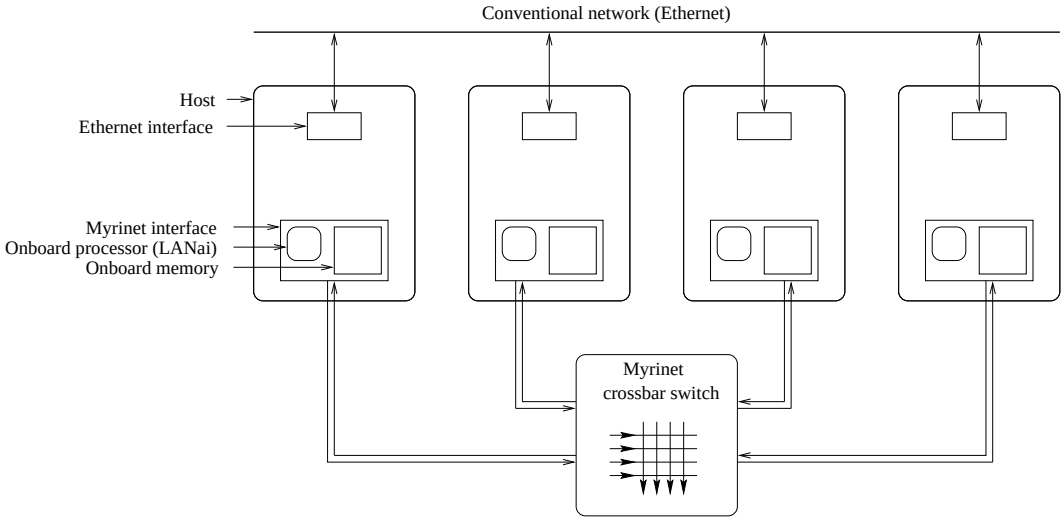

Figure 1: Myrinet on a Network of Workstations.

The switching network uses crossbar switches with a small number of ports (typically 4 to 16). The switches use cut-through routing and have a low latency (less than a microsecond). The switches and the interface boards are connected by fast (640 Mbit/sec) reliable links. Arbitrary interconnection topologies can be created in this way.

From a programming point of view, the interface boards are the more interesting and visible part of Myrinet. The board communicates with the network and with the host machine to which it is added. The board contains a fully programmable processor, called the *LANai*. The program running on this processor is called the *LCP* (LANai Control Program). The LCP can copy data to and from the host using DMA. Also, it can send data over the network and receive data from the network. The board contains a small amount of SRAM memory to store the LCP and to buffer incoming and outgoing data. The host can access this memory using either programmed I/O or DMA.

Various message passing libraries for Myrinet exist. For our research, we use the *Fast Messages* (FM) software from the University of Illinois [27]. FM includes a LANai Control Program and a library that is linked with the application running on the host. This library is used by the application programmer or a language RTS. It provides efficient and reliable point-to-point communication. Message reception in FM is based on polling: the application program periodically has to check if incoming messages are available. Calls to the FM send primitive also poll the network. FM maps the Myrinet interface board into user space and currently does not support sharing of this device between multiple applications.

3 Implementing the Orca System on Myrinet

In this section we discuss how the Orca system can be implemented on a Myrinet network. We first describe an initial design, which adheres to Panda's layered approach. Next, we use this design to illustrate the kind of performance problems that occur in a naive implementation of a high-level language on a high-speed network. Section 4 discusses how to address these problems.

3.1 Initial Design

Our initial design follows the layered approach outlined in Section 2.1. We have used this approach successfully for several other platforms, including networks of workstations and multicomputers. The overall structure is shown in Figure 2. The different software layers will be described below, starting at the lowest layer. The hardware we use is the Amoeba processor pool, which is a cluster of SPARC-based single-board computers connected by Ethernet. We have equipped eight pool processors with Myrinet interface boards, connected by an 8-port Myrinet crossbar switch.

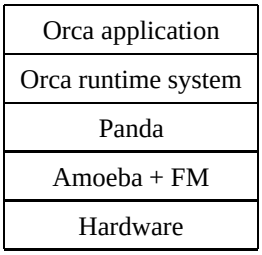

Figure 2: The structure of the Orca system.

Operating System Layer The lowest software layer is the interface to the hardware, as provided by the operating system or low-level communication library. In this case study, we use the Amoeba distributed operating system [32] extended with the Illinois Fast Messages (FM) software for Myrinet [27]. Amoeba provides kernel-level threads and communication facilities for the Ethernet. FM provides reliable point-to-point communication over Myrinet. As de-

scribed in Section 2.2, the Myrinet interface board is mapped into user space. The FM software runs partly on the host (in user space) and partly on the LANai processor on the interface board. The FM software that runs on the host takes care of writing message fragments to the interface board, reassembling received fragments out of the DMA area, and implementing credit-based flow control. The FM LCP is responsible for sending and receiving message fragments, and for depositing received fragments in the host's DMA area.

The Panda Layer The Panda layer implements RPC and totally-ordered multicast on top of FM. This layer was assembled from existing Panda implementations wherever possible, and custom modules were written for the parts that depend explicitly on FM. The standard Panda message module supports message fragmentation, because many operating systems have an upper bound on message size.

Message reception in Panda is based on an upcall model. FM, on the other hand, requires the application to poll the network for incoming messages. To implement the Panda upcall model on FM, we use a low-priority polling thread that continually polls the network whenever no other threads are runnable. Whenever a poll succeeds, FM makes an upcall to execute the message handler function. As with Active Messages [37], the handler function is passed as part of the message. In addition, the Orca compiler generates statements at the beginning of every procedure and loop entry to poll the network. The compiler already provided this functionality, since it is needed on other platforms (e.g., the CM-5) as well.

Since the FM library does not support multithreading (i.e., it is not thread safe), calls to the FM library are protected by a lock variable called the FM lock. To avoid deadlock during message handling, a thread that performs an upcall from a polling call to deliver a message may not hold any locks. The reason for this restriction is that the upcall handler function can also try to grab the lock. In particular, it could try to obtain the FM lock, because the upcall handler function is free to send messages (for which it needs the FM lock). Therefore, messages received from a polling call are queued (see Figure 3). They are handled either by a separate communication thread (if the poll is done from user code) or by the polling thread (after it has released the lock around the FM calls).

For the RPC implementation, a standard Panda module built on top of reliable point-to-point communication was available. This module handles arbitrarily sized messages by sending and receiving fixed-size fragments.

Multicast is supported neither by the Myrinet hardware nor by the FM software. A simple solution is to implement a spanning-tree protocol on top of FM point-to-point communication, which forwards a multicast message to each destination. This scheme provides reliable multicast since the FM point-to-point messages are reliable. With this scheme, however, messages from different sources may arrive in different orders at their destinations. Totally-ordered multicast requires that all messages (from all sources) be delivered in the same order everywhere.

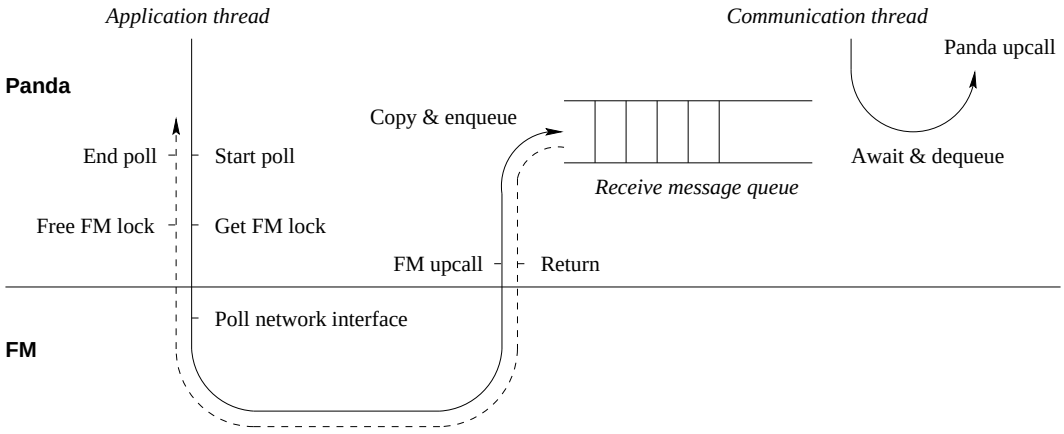

Figure 3: Upcall handling in the initial Myrinet implementation.

In our earlier research, we developed various reliable, totally-ordered, multicast protocols [13, 17]. The general idea is to use a central sequencer node that assigns a unique sequence number to each message. On switch-based networks, an efficient scheme is to have the sender fetch the next sequence number from the sequencer and then add this number to the message [13]. (Alternative schemes for unreliable, bus-based networks are described in [17]). Using the sequence numbers, Panda is able to delay out-of-order messages until their predecessors have also arrived. Higher layers (the Orca RTS) thus always receive messages in the right order.

Runtime System Layer The next layer in our system is the Orca RTS. The RTS is fully portable and only deals with the Panda communication primitives, so it does not depend on the underlying hardware or operating system. The RTS used in our initial Myrinet implementation therefore is the same as on all other platforms on which Orca runs.

The RTS implements nonreplicated objects using Panda RPC and replicated objects using Panda's totally-ordered multicasting. Also, the RTS takes care of marshalling and unmarshalling of data structures. In Orca, any data structure may be sent over the network (by passing it as a value parameter in an operation invocation). The routines to convert between Orca data structures and transmittable (flat) messages are provided by the RTS, using type-information generated by the compiler.

3.2 Performance Problems with the Initial Design

The Panda design strategy has been applied to many other platforms and has resulted in an efficient parallel programming environment for these platforms. On an Ethernet-based distributed system, for example, we obtain good performance at the Panda level (RPC and multicast), the Orca language level (object invocation), and the application level (speedups) [25].

Below, we study the performance of the Panda/Orca system on top of Myrinet, using the initial design outlined

	RPC	multicast	ROI	GOI
Myrinet	585	867	865	1170
Ethernet	1530	1580	1860	1960

Table 1: Communication latencies (in μsec) at the Panda and Orca level for the initial design.

above. The measurements were done on a system with eight SPARC processors running at 50 MHz. Each processor is on a single board (a Tatung board, equivalent to the Sun Classic) and contains 32 MB of local memory. The Myrinet boards we use contain the LANai 2.3 processor and 128 Kbyte of SRAM memory. The system runs the Amoeba 5.3 operating system extended with the FM 1.1 software. We give performance numbers both at the Panda level (RPC and multicast) and the Orca level (object invocations). The timings were obtained with a memory-mapped timer with a 0.5 μsec granularity.

The first two columns in Table 1 show the latency for empty messages using Panda RPC and totally-ordered multicast communication. The roundtrip latency for empty RPC messages (i.e., the null-latency) was measured by sending empty request and reply messages between two nodes. Multicast latency was measured using a group of eight members. Two of the members perform a ping-pong test (i.e., they multicast messages in turn), while the other six members only receive the messages. We report the average latency over all possible pairs of two different senders.

Likewise, we have measured the latency for Orca operations. ROI (RPC Object Invocation) uses a nonreplicated object containing a single integer, stored on one processor. Another processor repeatedly performs a (remote) increment operation on the object, using Panda RPC. GOI (Group Object Invocation) is a benchmark for operations on a replicated integer object. The object is replicated on eight machines. The latency is measured by having two machines in turn perform an increment operation on the replicated object. As for the multicast benchmark, we compute the average latency over all possible pairs of two senders. The latency for ROI and GOI is also shown in Table 1.

For comparison, we also show the null-latencies for the same benchmarks using an implementation of our system on top of Ethernet, using the same processors as the Myrinet implementation. It makes calls to Amoeba's low-level datagram protocol, FLIP [18], and uses the Panda communication protocols in user space [25]. As can be seen, the performance gain at the Orca level is approximately a factor of two, which does not even come close to the increase in network speed. Part of the performance gain is not even due to increased network speed, but is simply caused by the fact that Myrinet is mapped into user space, whereas the Ethernet device is accessed through calls to the Amoeba kernel.

The initial design has several performance problems. The latency for point-to-point messages (illustrated by the RPC and ROI measurements) is high. To illustrate, the roundtrip latency for 16-byte FM point-to-point messages over Myrinet is about 60 μsec on our hardware. The large overhead is mainly due to the handling of incoming messages. In

particular, since messages are received by a communication thread, a context switch at the client side is necessary to hand the reply message to the waiting thread. The latency for multicast communication also is high, because a multicast message is forwarded in software between different nodes. Also, our layered approach has some overhead. Finally, application programs will have an additional form of overhead: FM requires them to periodically poll the network for incoming messages. The overhead of polling is not reflected in the latency tests of Table 1.

4 Performance Optimizations

The initial design for implementing Orca on Myrinet presented above does not obtain good performance. The latencies achieved for the communication primitives are only slightly better than on a traditional Ethernet network. In this section, we discuss how to improve the performance. We describe several optimizations, some of which are already well-known. The goal of this section is to show which kinds of optimizations are necessary to obtain good performance on a high-speed network for a high-level language like Orca. The impact of the optimizations on benchmark and application performance will be discussed in Section 5.

The optimizations are based on three general ideas. First, we have implemented a user-level threads package that integrates message handling to reduce the overhead of polling and context switching. Second, we exploit the programmability of the network interface boards. By moving some critical parts of the Panda protocols from the host processor to the LANai processor, substantial performance gains can be obtained. Third, we use layer collapsing to reduce the overhead of Panda's layered approach.

4.1 Integrating Message Handling and Thread Management

The first problem we address is the software overhead of receiving incoming messages. On a SPARC (and many other RISC machines), the costs of receiving an interrupt and dispatching it to a user-level signal handler are already higher than the basic communication latency. So, generating an interrupt for an incoming message will drastically increase communication time. FM therefore does not use interrupts, but requires user programs to periodically check if a message has arrived (i.e., it uses polling).

Programmers using a high-level language should not be bothered with placing polling statements inside their programs. Consequently, the Orca compiler and RTS take care of polling automatically. This approach, however, leads to a performance problem since the compiler has to be conservative and inserts many polls to guarantee an acceptable polling rate. As we will see in Section 5.2.1, the overhead of automatic polling by the Orca system can be as high as 25%. In conclusion, the overhead introduced by polling may more than offset the savings of not getting an interrupt.

We have addressed this problem by integrating message reception and thread management. We use a user-level

threads package instead of Amoeba's kernel-level threads, so we can combine polling and interrupts. We have extended the FM LANai Control Program with the ability to generate an interrupt on message arrival. The program on the host can set a flag indicating whether or not it wants to get an interrupt from the LCP when a message arrives. Our threads package normally enables these interrupts, except when it enters the idle loop. If the application has no runnable threads, the thread scheduler disables message interrupts and starts polling until a message is available.

If a message arrives when the processor is doing useful work, we thus pay the price of an interrupt. Although interrupts are expensive, in our experience and that of others [16] the overhead of polling may be at least as high. In the worst case an interrupt is generated per incoming message, but it is also possible that multiple messages can be received with a single interrupt, given that a call to FM extracts all queued message fragments from the network interface. If the processor is idle when a message arrives it will be polling the network, so it immediately gets the message, without the overhead of an interrupt. This is important if a (single-threaded) application is waiting for a message, such as a server waiting for an RPC request, a client waiting for an RPC reply, or a process waiting in a barrier synchronization.

A second problem related to message receipt is the context switching overhead for handling messages. Incoming messages are handed to a separate communication thread in Panda (see Figure 3). In theory, the message could have been handled by the currently executing application thread. The problem, however, is that the handler function may block, which may result in deadlock (e.g., when the handler tries to lock an object that already is locked by the current thread). Doing a context switch to a separate thread is expensive compared to FM message latency. On our hardware, the cost of a context switch depends on the number of register windows used by the thread. For Amoeba kernel threads used with the initial Myrinet system, a context switch takes between 175 and 250 μsec; for the user-level threads, the costs are 45 μsec or higher.

By using user-level threads instead of kernel threads, we can thus save 130 μsec per context switch. These costs, however, can be reduced further with an optimization in the user-level threads package. The idea is as follows. When a message arrives, the Amoeba kernel interrupts the current thread and invokes a signal handler on the current stack as usual. Control transfers to the threads package, which creates a lightweight thread to execute the handler function. Instead of doing a context switch to the new thread, however, the threads package merely installs the new stack pointer and directly calls the handler function, just like a normal procedure call. This optimization is possible since the handler thread has no context that needs to be restored. Figure 4 shows the situation after the creation and invocation of the handler thread.

If the handler thread runs to completion without blocking, control automatically returns to the threads package in the signal frame, which clears the lightweight thread, restores the stack pointer, and resumes execution of the original

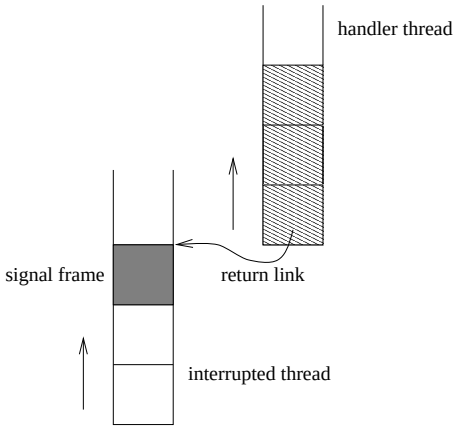

Figure 4: Fast thread allocation for message handling.

thread. If, on the other hand, the handler thread blocks, the lightweight handler thread is promoted to a full-blown thread by modifying the call stack of the handler. By changing the last return link in the call stack to point to a special exit function, the handler thread will not return to the code that resumes the blocked thread, but clean up itself instead. This allows the original thread to continue execution independently. In addition to upgrading the handler thread, the blocked thread is made runnable again so it will eventually resume execution.

This solution effectively solves the problem of context switching overhead. The overhead of allocating a lightweight thread is low, while the overhead of upgrading the handler thread has to be paid only in exceptional cases when the handler blocks. In the Orca system, this mainly occurs if the handler contains an operation invocation on an object and some local Orca process is currently also executing an operation on the same object. Since operations typically are short-lived in Orca, this is not likely to happen often.

Both problems discussed above (polling or interrupt overhead and context switching overhead) are certainly not unique to Myrinet. They are equally important on other platforms that have fast communication, such as the CM-5 [15]. On slower networks, the problems are less serious, because the relative overhead of an interrupt or context switch is much lower.

4.2 Exploiting the Programmability of the Interface Boards

Another performance problem in our initial design is the high cost of the Panda multicast protocol. Myrinet does not support multicasting in hardware, so Panda implements it in software. Even if an efficient spanning-tree algorithm is used, the cost of receiving and forwarding a message on each node is high. Another problem with the multicast protocol is that it requires a sequencer to order the messages. Interacting with the sequencer further increases the communication overhead.

We address these problems by exploiting the programmability of the Myrinet interface boards. Recall that each interface contains a processor (the LANai) on which a control program (the LCP) runs. The optimizations described below make use of the fact that this control program can be modified.

Most importantly, this flexibility allows us to implement the entire spanning-tree forwarding protocol on the interface processors, instead of on the hosts. Since the host processors are no longer involved in forwarding messages, the latency for a multicast is decreased substantially. With the new scheme, when the interface processor receives a multicast message from its host, it forwards the message to other interface processors, using a binary tree. When an interface processor receives a forwarded message, it likewise forwards it to other interface processors, and it also passes the message to its host (using interrupts or polling).

The protocol is illustrated in Figure 5. In this example, the main CPU on the first processor wants to multicast a message, so it passes this message to its LANai processor. The LANai sends the message to the LANai processors on the second and third host. The second LANai forwards the message to the fourth node. Each LANai receiving the message also DMAs it to a reserved memory area on its own host, from which it is retrieved by the application.

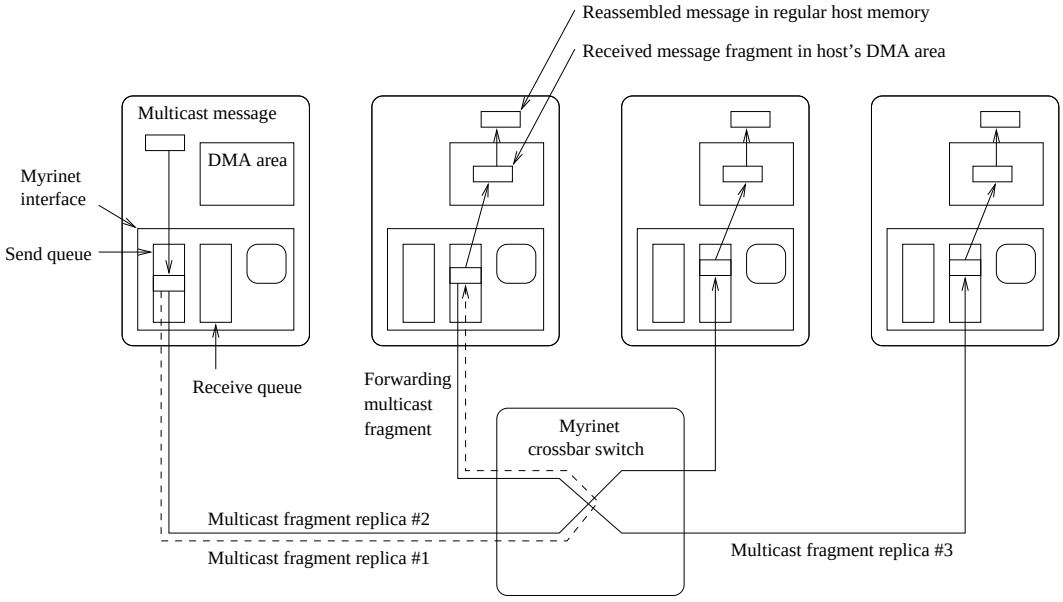

Figure 5: Multicast on the interface boards.

The basic problem with such a spanning-tree multicast protocol, however, is that it is unreliable. Even though the communication links are reliable, the protocol may drop messages. The key problem is that buffer space at the destinations will always be limited. If a message is multicast while one or more hosts are out of buffer space, these hosts will have to drop the message. This problem did not arise with the initial spanning-tree protocol on top of FM, because FM

uses a flow control mechanism for point-to-point messages [27]. The mechanism uses a credit-based scheme, where a sender is blocked if no buffer space is available for it at the receiver. In this way, FM never has to drop point-to-point messages and thus is reliable.

Our new protocol runs on the interface boards, rather than on top of FM, so it still may drop messages. The problem can be solved using a flow control mechanism that prevents buffer overflow, just as for FM point-to-point messages. However, flow control for multicast is a much harder problem than for point-to-point communication, because it must guarantee that buffer space is available at all nodes, instead of just one.

We have designed and implemented a flow control scheme for Myrinet [34]. Like the FM scheme, it is based on *credits*. One credit represents buffer space for one (fixed-size) message fragment at all nodes. A node can only multicast a fragment if it has a credit. Each node is given a number of credits in advance, so usually a node can start multicasting immediately. If a node runs out of credits, it must obtain new credits from a central credit manager. If a node has finished processing an incoming message, it should somehow inform the credit manager that its free buffer space has increased. To efficiently implement this recycling of credits we use a simple and efficient rotating token protocol.

The entire flow control scheme is implemented on the interface processors. The credit manager runs on one of the interface boards. Also, requesting new credits and recycling credits are completely implemented on the interface processors, without involving the host processors. As a result, we have succeeded in making the low-level multicast primitive reliable. The credit scheme guarantees that messages never have to be dropped due to lack of buffer space. The communication links are reliable, so messages are never dropped by the hardware either.

A final important optimization concerns the total ordering of multicast messages. The Panda protocol uses a central sequencer to order all multicast messages. Before a node multicasts a message, it first obtains the next sequence number from the sequencer. On Myrinet, we can optimize this scheme by managing the sequence numbers on the interface processors. The sequencer is run on one of the interface processors (rather than on a host), which again reduces message latency. Also, we use the same interface processor for both the sequencer and credit manager. The advantage is that a request for credits can now be piggybacked on the request for the next sequence number, thus reducing the overhead of credit requests.

We thus exploit the interface processors to speed up the forwarding of multicast messages and the requests for sequence numbers, and to run a credit scheme that makes multicasting reliable. We have implemented this functionality by modifying the LANai control program used by Fast Messages. The optimizations described in this section thus depend on the programmability of the interface processors. To summarize, the extensions we made to the original FM LCP are: to (optionally) generate an interrupt on message arrival; the implementation of a spanning tree forwarding

protocol; the implementation of a reliable credit-based multicast protocol; and the addition of an efficient primitive to obtain a sequence number. Several other networks also have programmable interface processors [7], so some of the optimizations may be applicable on such networks as well.

4.3 Layer Collapsing

The layered structure of our system also adds some overhead. Since FM and Panda are two independent systems, they each maintain their own pool of buffers, so they both have the overhead of buffer management. Also, both of them contain code for message fragmentation and reassembly. Finally, the initial design suffers from various runtime overheads. We discuss these issues in turn below. Many of these optimizations are not specific for high-speed networks. Still, the optimizations were necessary to obtain a good performance on Myrinet. We did not have to implement them for the original Ethernet-based system, because the performance gain would be relatively small compared to the communication time over the network.

We will first consider buffer management in FM and Panda. In the initial design, the FM layer communicates with Panda by means of an upcall with two arguments: a pointer to the buffer containing the received message and the size of the message. This message must be processed completely by Panda before returning from the upcall, since FM frees (or reuses) the buffer when the upcall returns. However, the Orca RTS does not guarantee that it handles incoming messages immediately. Therefore, Panda copies the message supplied by FM and queues this copy for processing by the RTS later on. This extra copy step clearly increases the latency.

To solve this problem, we changed the upcall interface of FM. In the new version, FM supplies the message in the form of a structure that may be retained by the higher layers as long as required. It is freed by an explicit call to a new FM primitive after it has been processed completely. Also, the FM message structure was extended with enough space (besides the data itself) to allow the higher layers to deal with the message (e.g., putting it in a queue) without having to allocate and use additional data structures. Clearly, this optimization gives up the clean separation between FM and Panda, so it is a form of layer collapsing.

Another problem with our initial design is the overhead of fragmentation. The initial design uses an existing message passing module that supports message fragmentation (because many operating systems have a maximum message size). Large messages are fragmented by Panda into chunks of 8Kbyte, and these chunks are fragmented again by the FM layer. FM, however, supports messages of arbitrary size. In the optimized system, we therefore use a custom Panda message module that dispenses with fragmentation.

Another form of overhead in the initial design concerns marshalling. In Orca, any data structure can be sent in a message over the network. Marshalling and unmarshalling of data structures is done automatically by the language

implementation. In the initial design, the compiler generated a *descriptor* for each operation, describing the types of the arguments of the operation. The RTS contained generic marshalling (and unmarshalling) routines that used these descriptors. These generic routines were quite expensive, since they had to determine the types of the arguments at runtime (using the descriptors). On an Ethernet, the relative overhead of marshalling is low but on Myrinet marshalling turned out to be relatively expensive.

We optimized marshalling by having the compiler generate a specific marshalling routine for each operation. For example, if an operation takes two arrays of integers as arguments, the compiler generates a routine that marshalls exactly these two arrays into a message (plus the inverse routine for unmarshalling). This optimization avoids the runtime interpretation of descriptors at the cost of increasing code size (for the marshalling/unmarshalling routines). However, it required significant extensions to the compiler for generating these routines. In the initial design, the compiler was not involved in marshalling at all (except for generating the descriptors), so this is again a form of layer collapsing.

We also made several performance improvements to the Orca RTS. We decreased the overhead of synchronization (by using fewer, more coarse-grained locks), message headers (by coalescing headers from different modules), and runtime statistics (which are used for decisions about object placement). Finally, we made various local changes to the code of each layer. In particular, the initial system suffered from procedure calling overhead. The SPARC processors have a fixed number of register windows. If the window overflows or underflows, an expensive trap into the operating system is made. For software engineering reasons, the initial system used many small procedures and thus suffered from window overflows and underflows. We solved this problem by inlining small procedures, mainly by using compiler directives. Furthermore, the GCC compiler supports an option (–mflat) to compile functions to explicitly save and restore registers on the stack instead of using register windows. By selectively compiling functions on the critical communication paths (i.e., the Panda RPC, multicast, and threads packages) with this option, the number of traps was reduced considerably without the need to restructure our software layering.

5 Performance

We will now study the impact of the optimizations on the performance of our system. We first look at low-level performance and describe the latency and throughput of some basic primitives in FM, Panda, and Orca. Next, we look at the performance of Orca applications, to see which improvements on speedups can be obtained using faster networks in combination with more efficient software. To measure the effect of the optimizations we have built two Orca systems:

- The initial design described in Section 3.1.
- The final system with the optimizations described in Section 4.

Optimization	Layer
Thread management	
Interrupts and polling	LCP/FM/Panda
Context switching	Panda
Programming the interface boards	
Totally-ordered multicast	LCP/FM
Layer collapsing	
Buffer management	FM/Panda
Fragmentation	FM/Panda
Marshalling	Compiler
Various overheads	Panda/RTS

Table 2: Overview of the performance optimizations.

To ease the discussion, Table 2 gives an overview of the optimizations described in the previous section. It also indicates at which layers the optimizations are implemented. (Here, FM denotes the part of FM running on the host, whereas the LCP runs on the interface board.)

5.1 Low-level Performance

We first look at the performance of several low-level benchmark programs that measure the efficiency of FM, Panda, and Orca.

5.1.1 Performance of Fast Messages

We have made several changes to the FM software on the host and the interface board. Most importantly, we have extended the LCP (LANai Control Program) with support for multicasting. With the current Myrinet hardware, such extensions easily decrease the performance of the low-level FM communication primitives. The reason is that the LANai processor on the interface board is very slow compared to the SPARC host CPU. The version of the LANai we use is a 5 MIPS 16-bit CISC processor. Even if the LCP only has to execute a few extra instructions, this immediately reduces performance [27].

With our modifications to FM, the LANai now has to examine every message to see if it is a multicast message that needs to be forwarded. Allocation of messages (on the host) and support for thread safety also add to the latency. Therefore, the point-to-point latency in our extended FM system will be higher than with the original FM. With the original FM code we achieved a minimal round-trip latency of 60 μsec for 16-byte messages on our hardware. The maximum throughput we achieve for messages of 16Kbyte is 8.6 Mbyte/sec (including reassembly of message fragments at the receiving host).

Our extensions have led to the following increases in round-trip latency. Calls to a mutex library to make the primitives thread safe add 7 μsec to the latency. Multicast-related changes to the LCP result in an additional 5 μsec over-

head. To increase the maximum achievable throughput we doubled the FM fragment size to 256 bytes, resulting in a sustained throughput of 10.1 Mbyte/sec for 16 Kbyte messages. This increased the roundtrip point-to-point time for 16-byte messages with another 10 μsec. Finally, the more flexible fragmentation code (including allocating and copying the message) adds another 10 μsec, resulting in a minimal roundtrip point-to-point latency of 60+7+5+10+10=92 μsec in total.

The latency for 16-byte messages thus has increased from 60 μsec to 92 μsec. For 64-byte messages the round-trip latencies have increased from 66 μsec to 103 μsec. For 248-byte messages (i.e., the maximum size for a message with one fragment), the latencies have decreased from 167 μsec to 143 μsec, because the original FM system sends two fragments in this case.

As discussed in Section 4, our optimized system uses one of the interface boards as a sequencer and credit manager for the totally-ordered multicast protocol. The time needed to fetch a sequence number and credits from the manager is 70 μsec.

In conclusion, our extensions add some overhead to FM. As we will see below, however, these extensions allow optimizations at the higher levels (Panda and Orca) that result in a large overall performance gain. In addition, the overhead of our extensions will decrease as faster interface processors become available, such as the new LANai 3.2, which is a faster 32-bit pipelined RISC processor. We have experimented with prototypes of this hardware, using our modified version of FM with a 128-byte message fragment size. The round-trip latency for a 64-byte point-to-point message decreased from 103 to 65 μsec. The latency for retrieving a sequence number and credits dropped from 70 to 42 μsec.

5.1.2 Performance of the Panda Communication Primitives

The optimizations described in Section 4 improve the performance of the Panda communication primitives. In particular, the improved message handling, buffer management, and fragmentation optimize both RPC and multicast performance. The optimizations implemented on the interface boards only make multicasting more efficient, but not RPC.

Table 3 contains the latency for empty messages and the throughput for 1 Mbyte messages of the RPC and multicast primitives, using the two different Panda systems described above. The latencies were measured as described in Section 3.2. The RPC throughput was measured by sending request and reply messages of equal size (1 Mbyte) between a pair of nodes. The multicast throughput is measured by having one member in a group of eight sending messages as fast as possible to the other members.

Based on these measurements, we can determine the impact of the optimizations discussed in Section 4. We will discuss the results for the latency of RPC and multicast messages. The total gain in this case is 414 μsec for RPC

	Initial	Final
RPC latency (μsec)	585	171
Multicast latency (μsec)	867	273
RPC throughput (Mbyte/sec)	4.7	10.1
Multicast throughput (Mbyte/sec)	2.6	5.5

Table 3: Communication latencies and throughput for the Panda primitives using the two different systems.

messages and 594 μsec for multicast messages.

We will analyze the RPC performance in some detail. Since our extended version of FM has an overhead of 32 μsec for roundtrip point-to-point messages (see Section 5.1.1), the higher-layer optimizations together gain $414 + 32 = 446$ μsec for RPC. The new threads package reduces the polling overhead by dispensing with polling from user code, using interrupts instead. Polling is done from the idle loop only. This optimization only affects the performance of applications and not of the benchmarks: the benchmarks poll for incoming messages anyway, since they are synchronous in nature. The implementation of the new threads package saves a thread switch for each incoming RPC reply message. Instead of giving the message to a separate communication thread, it uses an efficient mechanism for allocating a new thread. This optimization saves the costs of an Amoeba thread switch, which is 175 to 250 μsec. The layer collapsing optimizations together thus save approximately 200 to 250 μsec for RPCs. We have determined that about 120 μsec of this can be attributed to the removal of fragmentation code from Panda; the remainder is gained by simplified buffer management and by selectively having GCC generate code that does not use register windows.

The RPC latency for the final system is 79 μsec higher than the roundtrip latency for our extended version of FM (171 against 92 μsec). Unlike FM, however, the RPC protocol supports multithreading. In particular, when the client machine receives the reply message, the protocol wakes up the client thread that sent the request message, using a signal on a condition variable. FM, other the hand, does not support multithreading, so it does not have this overhead. Also, the Panda communication protocol needs to do demultiplexing between multicast and RPC messages, which also has some overhead. The throughput of the RPC protocol is the same as for FM, although the RPC protocol needs much larger messages to obtain a high throughput.

The performance of multicasting is improved by the same optimizations as described above for RPC, and by implementing the multicast protocol on the interface boards. To study the impact of the latter optimization, we have also built a Panda system that performs all optimizations discussed in Section 4, except for the multicast optimization. On this system, the multicast latency is 388 μsec, so the multicast optimization saves 115 μsec for empty messages. The multicast throughput on this system is 2.8 Mbyte/sec, so the multicast optimization also improves the throughput substantially.

	Initial	Final
ROI latency (μsec)	865	328
GOI latency (μsec)	1170	379
ROI throughput (Mbyte/sec)	3.2	5.5
GOI throughput (Mbyte/sec)	2.3	5.0

Table 4: Communication latencies and throughput for the Orca primitives using the two different systems.

5.1.3 Performance of the Orca Primitives

We now discuss the performance improvements of the Orca object invocation primitives. We measured the latency of the ROI (RPC Object Invocation) and GOI (Group Object Invocation) benchmarks described in Section 3.2. In addition, we measured the throughput of Orca operations using similar programs with a 1 Mbyte array as parameter. For ROI, an array of the same size is returned as the result of the operation. The ROI and GOI benchmarks thus have the same communication behavior as the RPC and multicast benchmarks discussed above, except that they also suffer from any overhead introduced by the Orca RTS (such as marshalling and locking). The results are shown in Table 4.

The performance gains for ROI and GOI can largely be attributed to the improvements of the Panda RPC and multicast primitives, discussed above. In addition, performance has been improved by having the compiler generate specialized marshalling routines. We have measured that this saves about 60 μsec for the ROI benchmark. Finally, we have made several local improvements to the RTS, which also reduce the latency. We can determine the overhead of the Orca RTS for the final system by comparing Tables 3 and 4. For ROI, the overhead is 157 μsec and for GOI it is 106 μsec.

The throughput for ROI is worse than for RPC (5.5 Mbyte/sec against 10.1 Mbyte/sec). The reason is that the Orca compiler generates code that copies the data several times. The input parameter and the result each are copied twice, during marshalling and unmarshalling. The number of copies could be reduced through compiler optimizations, but our current system does not do this yet.

5.2 Performance of Orca Applications

To evaluate whether or not the investment both in hardware and software for faster networks really improves the performance of parallel applications, we have used two Orca applications to compare their performance on Ethernet and the two Myrinet implementations described in the previous sections. One application mainly uses point-to-point (RPC) communication, whereas the other is dominated by multicast communication. Given the differences in performance for the basic Orca operation invocation mechanisms (i.e., ROI and GOI), we expect that fine-grained Orca applications that invoke many operations on shared objects will benefit considerably from the lower latencies on Myrinet. Also, applications that require high bandwidths will benefit from the crossbar interconnection switch of Myrinet, which fa-

cilitates parallel data exchange between pairs of nodes. On traditional Ethernet, all communication is serialized, which degrades performance of applications that communicate in parallel.

5.2.1 Water

The first application we discuss is the Water application from the Splash benchmark suite [30]. We have rewritten this program in Orca and use it to simulate the behavior of a system with 100 water molecules for 100 time steps. The molecules are statically distributed over the parallel processors. Each processor has one object that contains information about its molecules. At the beginning of each time step, each process fetches the updated positions of certain molecules from remote objects, using remote object invocations. The positions are used to compute the inter-molecular forces and new positions for the molecules. Since the processes are running in lock-step, all the position updates have to be fetched at the same time, which makes Water a communication intensive application for small numbers of molecules.

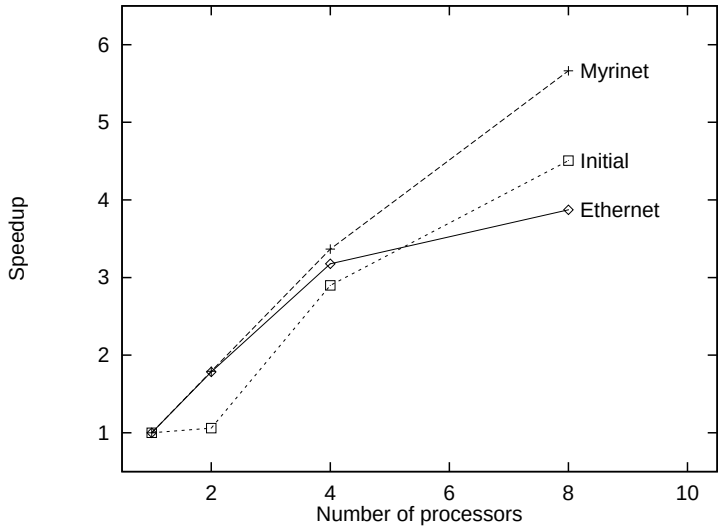

Figure 6: Parallel performance of Water.

Figure 6 shows the speedups of the Water program on the three different systems, relative to the time on one processor for the final Myrinet system. The performance on Ethernet shows that Water does not achieve a good speedup for this problem size on eight processors. The communication generated by the Water program cannot be handled fast enough by the Ethernet, hence the efficiency drops below 50%. On Myrinet, the lower latency and increased bandwidth of the network have a strong impact on performance. Water achieves a speedup of 5.7 on eight processors, using our final system. Thus for Water, the usage of a high-speed network improves performance considerably.

Figure 6 also shows that the many optimizations of the Orca system software are of great importance, since the speedup curve of the initial design is much lower than the one for the final Myrinet system. The weak performance of

the initial design is mainly caused by the need to poll the network explicitly from the Orca code. The Orca compiler conservatively inserts polling statements at each function entrance and loop body. To save on the number of network polls, the generated code decrements a global counter and only invokes a network poll when the counter reaches zero. We have measured that the polling overhead can be very high: the sequential execution time of Water increases 25% when polling statements are included.

This high overhead is not solely caused by the additional polling instructions. Certain optimizations in the C compiler also become unapplicable. For example, the C compiler for the SPARC optimizes leaf functions and avoids using a register window. Leaf functions that now contain a function call (to poll the network) can no longer be optimized in this way, which introduces additional traps to the operating system. Also, the C compiler performs less function inlining, because of the larger code fragments and additional function calls to poll the network.

The poor performance of Water on two processors using the initial design is mainly caused by polling overhead. The single processor case was compiled without any polling statements, while the two processor case necessarily contains polling statements. We tried to tune the polling frequency by only invoking a real network poll once every N compiler generated polling statements, but this has proven to be difficult. The measurements presented were obtained with $N = 1000$. Increasing the frequency to $N = 100$ raises the polling overhead to a factor of two, because of the additional network polls. Decreasing the rate to $N = 10000$ also has a negative impact on performance because the network is not polled fast enough. Setting the polling frequency right is a difficult problem, as recognized by others [15]. The optimized system handles incoming messages through interrupts, which is much more efficient for Water. We have determined that the overhead of such interrupts on execution time is less than 0.5% (using eight processors).

Comparing the speedups of Water on Ethernet and the initial Myrinet implementation shows that for this application the usage of a faster network pays for larger numbers of processors, but that a naive port of high-level software does not always pay off. The Ethernet implementation outperforms the initial Myrinet implementation on two and four processors.

5.2.2 Linear Equation Solver

The second application we discuss is a linear equation solver using the Jacobi method. The Orca program is set up to solve a system of 300 equations, and it needs 2070 iterations to compute a solution. At the end of each iteration, the Orca processes synchronize and exchange data through a single shared object. This object is being replicated by the Orca runtime system so that all processes can read the updated state from the local copy without any communication. Thus at each iteration, N Orca processes effectively multicast their local update, which result in N multicast messages. Since the Orca processes are running in lock step, all communication occurs roughly at the same time.

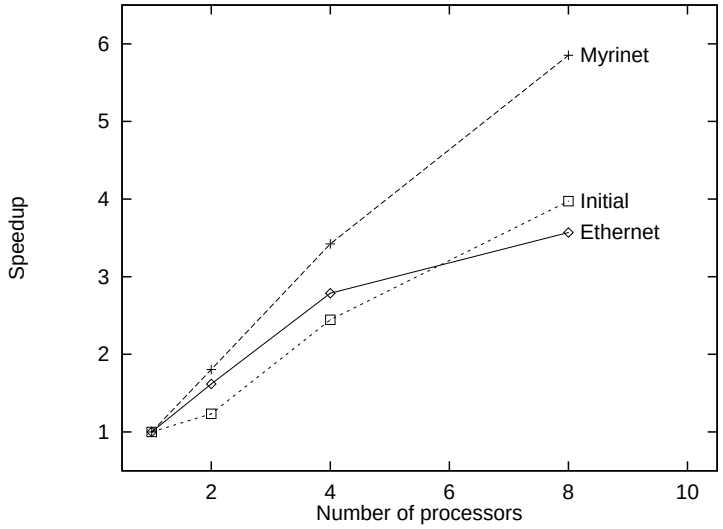

Figure 7: Parallel performance of the Linear Equation Solver.

The performance of the linear equation solver (see Figure 7) shows that the Ethernet implementation does not achieve satisfactory speedup on eight processors because the bus based network cannot handle the peak load of multicast communication. The Myrinet implementation, however, does much better and reaches a speedup of 5.9 on eight processors. The crossbar network topology of Myrinet allows forwarding of the multicast messages to happen in parallel, which (together with the lower latency of multicasting) explains the better performance results.

The speedup curve of the linear equation solver on the initial Myrinet design shows a similar behavior as for the Water application. Performance is severely degraded because of the polling overhead, but for a larger number of processors the underlying Myrinet network outperforms the Ethernet. Again the ability to communicate in parallel has proven to be a great advantage of the Myrinet network.

6 Related Work

In this section we compare our work with that of related systems. Several other systems exist that (like FM) try to obtain very high performance for low-level primitives on high-speed networks, using either a message passing or shared-memory programming model [1, 24, 33, 35, 38]. We discuss one such system, U-Net, in more detail. U-Net [36] is a user-level network interface on top of which higher-level protocols (e.g., TCP/IP, UDP/IP) and languages (e.g., Split-C) have been built. An implementation of U-Net over ATM exists that exploits the programmability of the interface board (Fore Systems SBA-200) to obtain high performance. An important issue addressed by U-Net is protection. U-Net multiplexes the network interface among multiple processes and gives each process protected access to the network. To

implement this with user-space protocols, they again exploit the programmability of the interface and use its firmware to enforce protection. Neither FM nor our work currently supports protection. Instead, we allow only one application to access the network device at the same time.

In addition to these low-level primitives, several higher-level abstractions have been implemented on high-speed networks. Many of these implementations are in the form of libraries rather than languages. CRL [16] is a highly efficient, language-independent, user-level distributed shared memory. CRL supports shared regions, which are similar to Orca's shared objects, except that they are accessed through library calls. CRL has been implemented on the CM-5 and Alewife.

Xu and Fisher [40] describe an implementation of PVM on top of Myrinet. The original PVM system uses TCP/IP and UDP/IP. This paper describes how to use a fast data path that avoids these protocols and by using Myrinet's API (Application Programming Interface) for data traffic. In this way, they obtain a high throughput for large messages. The latency for small messages is still high (about 600 μsec). For Orca we obtain a much lower latency, partly because of the optimizations described above and partly because we use FM instead of the much slower API. Scales et al. [28] describe a reimplementation of the PVM interface on an AN2 ATM network. A high performance is obtained by changing the AN2 firmware, the operating system, and the PVM library.

Several other papers discuss the implementation of multicast communication on modern networks. Huang and McKinley [14], for example, describe how to implement multicasting (and other collective communication operations) on ATM, on top of either PVM or AAL5. Gerla et al. [8] discuss possible hardware and software implementations of multicast on Myrinet. The hardware solution proposes a modification of the Myrinet switches, to let them take care of duplicating the multicast worms to multiple destinations, without the possibility of deadlock. For the software implementation of multicast, they propose an alternative to our buffer reservation scheme. Buffer deadlock (which cannot occur in our system) is avoided by means of two buffer classes, one for sending to nodes with a higher "node-id", the other for the reverse direction. Furthermore, rather than allocating buffer space for a multicast message at all destinations, the paper proposes an optimistic protocol with acknowledgements and possible retransmission in case of buffer overflow. Comparing the performance of Gerla's software solution with our approach is difficult, since the prototype discussed in [8] is based on the Myrinet API, which has a much higher latency than FM. Also, our credit scheme contains a caching mechanism so that for small multicast messages the credit retrieval is not in the critical path. In practice our "conservative" solution does not introduce much extra delay when compared to an "optimistic" solution. Moreover, the "optimistic" scheme also introduces acknowledgement related overhead on the LCP and the network.

In their investigation of the sources of software overhead in messaging layers, Karamcheti and Chien [20] found that

messaging layers incur significant software overheads when implementing "high-level" features not provided by the network hardware (e.g., FIFO message ordering, deadlock safety, reliable delivery, and buffer management). We too have found that high-level services incur large software overheads due to limitations of both the network hardware (no flow control, no multicast) *and* the low-level messaging software (not thread safe, no interrupt-based message delivery, handler restrictions). To reduce these overheads, many optimizations in the low-level software as well as the higher-level RTS were needed.

The optimizations in our threads package to use an inexpensive mechanism for creating and terminating message handler threads resemble the thread optimizations of Stacklets [9] and Lazy task creation [23]. Their goal, however, is to reduce the overhead of fine-grained parallel programs using a large number of threads. The key idea is to inline the execution of child threads and upgrade the child to a full thread only in the case of blocking, just like with our message handler threads. The fine-grained nature of their parallel programs, however, forces them to modify the compiler to use a slightly more expensive calling sequence. In our case, the thread package can swap the stack pointers without compiler support. Another difference is that a child thread needs to synchronize with its parent, while our handler threads can simply run to completion once upgraded to a full blown thread. This also makes our model easier to implement.

Several researchers have recognized that both polling and interrupts are useful mechanisms to extract messages from the network [5, 6, 22, 36]. Polling is cheaper than taking an interrupt and makes sense when a message is expected to arrive soon (e.g., the reply of an RPC). In addition, with explicit polling the programmer controls when exactly message handlers are run. This control can be used to achieve mutual exclusion between message handlers and computational threads. Interrupts, on the other hand, suit communication patterns in which messages arrive asynchronously. Remote Queues [5] stress the benefits of polling, but do provide *selective* interrupts; polling is used by default and interrupts are only generated for specific kinds of messages (e.g., system messages). With Orca, it is harder to fully exploit polling since the RTS hides all communication from the programmer and usually cannot predict when a message will arrive. Therefore, we use polling only when all computational threads are idle (i.e., when polling imposes no overhead) and use interrupts in all other cases. Maquelin et al. [22] have proposed hardware that generates an interrupt only when the network has not been polled for a while. We could use a similar mechanism by modifying the FM LCP to delay the generation of interrupts.

7 Discussion

We have implemented a high-level parallel programming system (Orca) on a high-speed network (Myrinet). The programming system provides a form of object-based distributed shared memory to the user. It is implemented with a

portable system that uses a layered approach. To implement this system efficiently on a high-speed network, many optimizations are required that are not necessary (or even possible) on a traditional LAN such as Ethernet. We have made two optimizations to message handling, to reduce the polling and context switching overhead. We have implemented an efficient reliable multicast protocol (which is used to implement replicated objects) using the programmability of the network interface boards. Finally, we have implemented various forms of layer collapsing.

The paper gives a detailed analysis of the performance of our system. The latency of remote object invocations in Orca on Myrinet is 328 μsec. This is a substantial improvement over a traditional Ethernet-based implementation, where the latency is 1860 μsec. The latency for updating a replicated object has decreased from 1960 μsec for Ethernet to 379 μsec for Myrinet. Without the optimizations mentioned above, the latencies are roughly a factor of two better on Myrinet than on Ethernet. Also, the performance of applications has been improved substantially by using Myrinet in combination with the optimized Orca system.

The most important optimizations in our system are: to save a context switch for message handling; to do fragmentation only at the lowest level (FM) and not at the Panda level; and to implement multicasting in firmware on the interface boards. Each of these optimizations saves over 100 μsec on the latency of object operations. Having the compiler generate specialized marshalling routines saves 60 μsec. Also, using interrupts instead of compiler-generated polls substantially improves the performance of applications.

Compared to lower-level message passing primitives, Orca still has a substantial overhead, which is due to its much higher level of abstraction. For example, the latency of remote object invocation is about a factor of five higher than the roundtrip latency for FM messages on the same hardware. The functionality provided by the Orca primitives, however, is much higher. To study the differences between low-level and high-level programming systems, we discuss the differences between FM and Orca in some more detail.

FM provides efficient, low-level send/receive primitives with very little functionality. Higher-level abstractions can be built on top of FM. The FM user can send a message to a certain machine using `FM_send_4` (for messages up to four words) or `FM_send`. Messages are received by periodically calling `FM_extract`, which checks if messages are available. The send primitives have to supply their own buffers. Finally, the FM primitives are not thread safe.

The programming model provided by Orca is of a much higher abstraction level. Orca users are not aware of the various issues of physical communication, such as sending messages, polling, buffer management, and so on. Instead, they only deal with two simple language constructs: processes and shared objects. The shared objects provide a form of distributed shared memory, which in general is easier to program than message passing. For example, if an application needs to maintain some global state information, this state can simply be stored in a shared object. The Orca compiler

and RTS will decide how to implement the object. The object may be replicated automatically on several machines, with the RTS taking care of coherency. With message passing, it is much more difficult to implement such shared information. Another difference between FM and Orca is that marshalling is done automatically in Orca. The FM send and receive primitives only accept contiguous buffers. If a more complicated data structure has to be transferred, the user must provide the marshalling and unmarshalling routines. In Orca, all of this is done transparently by the compiler and RTS, without any involvement from the programmer.

The goal of this paper is not to show the ease of use of Orca. (A detailed evaluation of the usability of Orca can be found elsewhere [39].) However, the level of abstraction of a programming system clearly has implications for the implementation and performance, especially when using a high-speed network like Myrinet. The FM primitives have a low latency on Myrinet, so any overhead in the language implementation affects performance.

During the case study described in this paper we have learned many lessons that also will be applicable to other high-level languages. Below, we summarize the most important lessons.

Many high-level parallel languages support multithreading and hide low-level issues like polling and marshalling from the programmer. In our experience, the performance overhead of multithreading can be high if implemented naively, since delivering an incoming message at the right thread is expensive. We invested much effort in reducing the context switching overhead. This optimization is successful, but we had to switch to a new, user-level threads package and we had to make complicated low-level modifications to the internals of the package.

Our system also succeeds in freeing the programmer from the burden of polling the network. Our solution is based on a simple and general idea (poll when the processor is idle, use interrupts otherwise), but to implement this we again had to resort to low-level systems programming, including modifications to the firmware and the threads package. Marshalling of data structures can easily be handled by the compiler, although the overhead is significant. We reduced the overhead by generating specialized marshalling routines, but still the copying overhead of the marshalling routines is high and limits throughput. Further optimizations are required to reduce this overhead.

Our work on the multicast optimizations is useful to many other programming systems, since multicasting is supported in an increasing number of systems. We have shown that the latency of multicast operations can be decreased substantially by implementing the spanning tree forwarding routine in firmware. A critical issue is the possibility of buffer overflow, but a proper flow-control scheme can adequately solve this problem. An important lesson is that a programmable interface processor gives many opportunities for important performance optimizations.

Another lesson from our work is that a layered approach to achieve portability is feasible even on a high-speed network, although some amount of layer collapsing is probably unavoidable to obtain good performance. In our system,

parts of one layer (Panda) have been integrated with the lower-level message passing layer (FM). The resulting system, however, still uses a layered approach. The runtime system, for example, still is a separate layer that only deals with objects and processes, and not with communication protocols.

In conclusion, the paper has shown that substantial performance gains are possible by implementing a high-level language on a fast network, but that such gains can only be achieved using extensive optimizations.

Acknowledgments

We thank Debby Wallach and Frans Kaashoek for the initial version of the threads package used in this paper, and Andrew Chien and Scott Pakin for making the FM software available to us. The Orca Water program was written by John Romein and Frank Seinstra. We are grateful to Greg Benson, Matt Haines, and Andy Tanenbaum for their useful comments on a draft of this paper.

References

[1] T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15:54–64, February 1995.

[2] H.E. Bal and M.F. Kaashoek. Object Distribution in Orca using Compile-Time and Run-Time Techniques. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 162–177, Washington D.C., September 1993.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

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

[5] E.A. Brewer, F.T. Chong, L.T. Liu, S.D. Sharma, and J.D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *ACM Symp. on Parallel Algorithms and Architectures '95*, Santa Barbara, CA, 1995.

[6] D. Chiou, B.S. Ang, Arvind, M.J. Beckerle, A. Boughton, R. Greiner, J.E. Hicks, and J.C. Hoe. StarT-NG: Delivering Seamless Parallel Computing. In *Proceedings of the First International EURO-PAR Conference*, pages 101–116, Stockholm, Sweden, August 1995.

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

[8] M. Gerla, P. Palnati, and S. Walton. Multicasting Protocols for High-Speed Wormhole-Routing Local Area Networks. Technical Report SSN Project Internal Note, UCLA, 1996.

[9] S.C. Goldstein, K.E. Schauser, and D. Culler. Enabling Primitives for Compiling Parallel Languages. In B.K. Szymanski and B. Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 153–168, Boston, MA, 1995. Kluwer Academic Publishers.

[10] A.S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[11] D.G. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(1):10–22, February 1992.

[12] M. Haines, B. Hess, P. Mehrotra, J. van Rosendale, and H. Zima. Runtime Support for Data Parallel Tasks. In *Proc. Frontiers 1995*, pages 432–439, January 1995.

[13] H.P. Heinzle, H.E. Bal, and K. Langendoen. Implementing Object-Based Distributed Shared Memory on Transputers. In *Transputer Applications and Systems '94*, pages 390–405. IOS Press, September 1994.

[14] C. Huang and P.K. McKinley. Communication Issues in Parallel Computing Across ATM Networks. *IEEE Parallel and Distributed Technology*, 2(4):73–86, Winter 1994.

[15] K.L. Johnson. High-Performance All-Software Distributed Shared Memory. Technical Report MIT/LCS/TR-674 (Ph.D. thesis), Massachusetts Institute of Technology, December 1995.

[16] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance All-Software Distributed Shared Memory. In *15th ACM Symp. on Operating Systems Principles*, pages 213–228, Copper Mountain, CO, December 1995.

[17] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, Amsterdam, December 1992.

[18] M.F. Kaashoek, R. van Renesse, H. van Staveren, and A.S. Tanenbaum. FLIP: an Internet Protocol for Supporting Distributed Systems. *ACM Transactions on Computer Systems*, 11(1):73–106, January 1993.

[19] L.V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 91–108, Washington D.C., September 1993.

[20] V. Karamcheti and A.A. Chien. Software Overhead in Messaging Layers: Where Does the Time Go? In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1994.

[21] J.W. Lee. Concord: Re-Thinking the Division of Labor in a Distributed Shared Memory System. Technical Report TR-93-12-05, Univ. of Washington, Seattle, WA, 1993.

[22] O. Maquelin, G.R. Gao, H.H.J. Hum, K.B. Theobald, and X. Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *The 23rd Annual International Symposium on Computer Architecture*, Philadelphia, Pennsylvania, May 1996.

[23] E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[24] S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, and J. Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 68–79, July 1995.

[25] M. Oey, K. Langendoen, and H.E. Bal. Comparing Kernel-Space and User-Space Communication Protocols on Amoeba. In *Proc. of the 15th International Conference on Distributed Computing Systems*, pages 238–245, Vancouver, British Columbia, Canada, May 1995.

[26] W.G. O'Farrell, F.Ch. Eigler, I. Kalas, and G.V. Wilson. ABC++ User Guide. Technical report, IBM Canada, Toronto, 1995.

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

[28] D.J. Scales, M. Burrows, and C.A. Thekkath. Experience with Parallel Computing on the AN2 Network. In *10th International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996.

[29] D.J. Scales and M.S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. *Proc. 1st Symp. on Operating System Design and Implementation*, pages 101–114, November 1994.

[30] J.P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *ACM Computer Architecture News*, 20(1):5–44, March 1992.

[31] A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal. Parallel Programming using Shared Objects and Broadcasting. *IEEE Computer*, 25(8):10–19, August 1992.

[32] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(2):46–63, December 1990.

[33] C.A. Thekkath. System Support for Efficient Network Communication. Technical Report 94-07-02 (Ph.D. thesis), Univ. of Washington, July 1994.

[34] K. Verstoep, K. Langendoen, and H.E. Bal. Efficient Reliable Multicast on Myrinet. Technical Report IR-399, Vrije Universiteit, Amsterdam, January 1996.

[35] T. von Eicken, A. Basu, and V. Buch. Low-Latency Communication over ATM Networks Using Active Messages. *IEEE Micro*, February 1995.

[36] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating System Principles*, pages 303–316, Copper Mountain, CO, December 1995.

[37] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *The 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[38] D.A. Wallach, W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, and W.E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 217–226, Santa Barbara, CA, July 1995.

[39] G.V. Wilson and H.E. Bal. An Empirical Assessment of the Usability of Orca Using the Cowichan Problems. *IEEE Parallel and Distributed Technology (to appear)*, 4, 1996.

[40] H. Xu and T.W. Fisher. Improving PVM Performance using ATOMIC User-Level Protocol. In *Proc. First Int. Workshop on High-Speed Network Computing*, pages 108–117, Santa Barbara, CA, April 1995.