Cluster Computing 2 (1999) 107–116

Efficient layering for high speed communication: the MPI over Fast Messages (FM) experience

Mario Lauria [a], Scott Pakin [b] and Andrew Chien [c]

[a] *Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114, USA*
[b] *Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, IL 61801, USA*
[c] *Science Applications International Corporation Chair Professor, Department of Computer Science and Engineering University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114, USA*

We describe our experience of designing, implementing, and evaluating two generations of high performance communication libraries, Fast Messages (FM) for Myrinet. In FM 1, we designed a simple interface and provided guarantees of reliable and in-order delivery, and flow control. While this was a significant improvement over previous systems, it was not enough. Layering MPI atop FM 1 showed that only about 35% of the FM 1 bandwidth could be delivered to higher level communication APIs. Our second generation communication layer, FM 2, addresses the identified problems, providing gather-scatter, interlayer scheduling, receiver flow control, as well as some convenient API features which simplify programming. FM 2 can deliver 55–95% to higher level APIs such as MPI. This is especially impressive as the absolute bandwidths delivered have increased over fourfold to 90 MB/s. We describe general issues encountered in matching two communication layers, and our solutions as embodied in FM 2.

1. Introduction

Dramatic advances in low-cost computing technology have combined to make clusters of PCs an attractive alternative to massively parallel processor (MPPs) architectures. Leveraging on mass-market volumes of production, the humble PC has benefitted from huge and ever increasing investments in the development of its key components (CPU, memory, disks, I/O buses, peripherals), while at the same time MPP manufacturers are coming to terms with a contraction of the market for multi-million dollar machines.

However a supercomputer is more than a collection of high performance computing nodes; it is in the way its component parts are integrated that the real challenge of a parallel machine design lies. In comparing a cluster architecture with the custom design of a contemporary MPP, it is in the interconnection technology that the latter has the largest edge over the former. For example, the Cray T3D achieves communication latencies of about 2 μs and a peak bandwidth of about 300 MB/s, the IBM SP2 of about 30 μs and 40 MB/s, respectively, whereas the typical values for a classical Ethernet-interconnected cluster are 1 ms and 1.2 MB/s, respectively.

The new high speed Local Area Networks (LANs) available today (ATM [4], FDDI [11], Fibrechannel [1], Myrinet [2]) offer comparable hardware latency and bandwidth to the proprietary interconnect found on MPPs. The introduction of these enabling technologies shifts the focus of the MPP versus cluster comparisons from performance to more general considerations of system scalability, reliability, affordability, and software availability.

New hardware technologies are only part of the communication picture; delivering performance to applications requires communication software capable of delivering the network performance. Fast network hardware alone is not sufficient to speed up communication [20]. Existing communication protocols have been developed to address requirements of robustness in the presence of unreliable transport and large network latencies, along with operating system controlled access to network interfaces. As a consequence, they are characterized by a large processing overhead, which prevents them from fully exploiting the performance of the new networks (figure 1).

Over the past few years, many research projects have studied the design of high performance communication software (Fast Messages (FM) [6,12,21], Active Messages (AM) [30], U-Net [31], VMMC-2 [10], PM [29], BIP [26]). In the Fast Messages project, we built two generations of systems optimized to deliver communication performance to the application. The first generation, FM 1.0, was based on our studies of essential communication guarantees (reliable, in-order communication with flow control) and tuned for realistic message-size distributions (mostly short messages). FM 1.0 achieved dramatically more usable communication performance, reducing the half-power message size

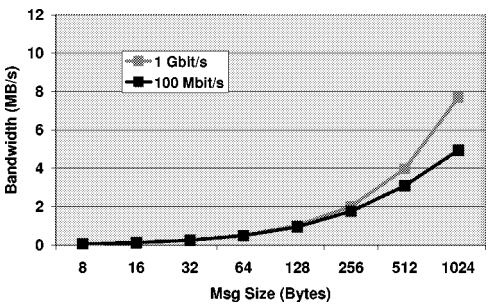


Figure 1. Theoretical bandwidth over 100 Mbit/s and 1 Gbit/s Ethernet assuming a fixed 125 μs protocol processing overhead.

for the Myrinet network by nearly two orders of magnitude, from over four thousand bytes to 54 bytes. We present the results of our initial experience with the implementation of user-level libraries on top of FM 1.x, which expose the critical issues and the important services required in matching two adjacent layers of the communication hierarchy.

For the second generation Fast Messages system, we used the insights gained from using FM 1.x to optimize the FM API and maximize the portion of FM performance delivered to the applications. By building high-level libraries such as MPI on top of FM and analyzing the resulting performance of the entire software stack, we found that a number of inefficiencies were created at the interface between libraries. The performance losses caused by the interface are remarkable, limiting network performance to a small fraction ($<$10%) of the hardware.

Fast Messages 2.x eliminates these interface problems, enabling over 90% of FM's performance to be delivered to higher level API's such as MPI. We describe the new elements of the FM 2.x API: gather/scatter, interlayer scheduling, receiver data pacing and their impact on usable performance. The interface efficiency obtained with the FM 2.x interface is over 80% for almost all message sizes, even for four byte messages, and peaks to 98%, a dramatic improvement. The implementation of MPI-FM atop the FM 2.x API achieves 91 MB/s peak bandwidth versus the 92 MB/s available on FM. The performance increase is even more impressive considering the more than fourfold increase of absolute performance of FM 2.x with respect to FM 1.x as a result of the migration from a Sparc to an x86 architecture.

The remainder of the paper is organized as follows. In section 2 we review the results that motivate the design of FM. In section 3 we present the FM 1.x API and discuss its strengths and weaknesses. In section 4 we present the FM 2.x API and describe its features. Related work is surveyed and contrasted with our work in section 5. Finally, we make a few concluding remarks in section 6.

2. Motivation for Fast Messages designs

The design of Fast Messages is motivated by the wealth of knowledge about message size distributions, the characteristics of traditional network protocols, and studies of high performance networks in parallel computers. The core of these results is summarized below.

2.1. Network traffic characteristics

From the first use of computer networks, scientists have studied the size, the frequency, and distributions of both for network traffic. Such studies consistently show that the majority of traffic (by packet count) consists of short messages. This property is remarkably stable across networks, time, and applications [13,18]. In a study of traffic on an Ethernet connecting diskless workstations to file servers [13], Gusella found that the majority of packets were less than 576 bytes; of these 60% were 50 bytes or less. In another study [18], Kay et al. measured the TCP and UDP traffic on a FDDI LAN of Unix workstations in a university computer science department. They found that TCP message sizes are small: over 99% of packets are less than 200 bytes. UDP traffic was slightly larger, with 86% of messages being less than 200 bytes. NFS-generated UDP packets accounted for 90% of the traffic measured. Continuous studies at the SUNY-Buffalo campus also chronicle the predominance of short messages. For a wide variety of networks, across a wide range of time, packet average sizes of 300–400 bytes were recorded.

The prevalence of short messages implies that if good network performance is to be accessible, it must be delivered to short messages. This is in contrast to many gigabit network projects that required megabyte-sized messages to deliver gigabit bandwidth. In short, overhead must be minimized, as at high network speeds, there is little spooling time available to mask network overhead.

2.2. Legacy protocols

Widely used Internet protocols such as TCP [25] and UDP [24] provide widespread interoperability and two levels of functionality – reliable byte streams and unreliable datagrams. However, these protocols incur significant overheads [8], essentially preventing the delivery of network performance to short messages. For example, the *fastest* implementations of UDP achieve per packet overheads of ≈ 125 μs. This implies that for typical packet size distributions ($<$256 bytes), bandwidths of no greater than 2 MB/s could be sustained. Of course, the overheads for reliable protocols such as TCP are even greater.

2.3. High performance communication layers

To identify crucial performance factors in high speed communication software, we undertook empirical studies of communication layers inside parallel computers. These studies identified the key guarantees a communication layer must provide to avoid incurring a large software overhead at higher levels of the system. Our study of CM-5 Active Messages (CMAM) [14] measured the dynamic instruction count of the CMAM assembly code and identified the overhead contributions of the range of guarantees provided by the communication layer (in-order delivery, buffer management, fault tolerance). Because the network of the CM-5 provided none of these features, the software overhead can be considered the "cost" of each feature on the CM-5. In a highly optimized messaging layer like the CMAM up to 50–70% of the software messaging costs are a direct consequence of the gap between user requirements such as in-order and reliable delivery, end-to-end flow control, and actual network features like arbitrary delivery order, finite buffering, unreliable communication. For example, in one case (16-word messages, 4-word packet size, multi-packet delivery) 216 out of a total of 397 cycles are spent for buffer

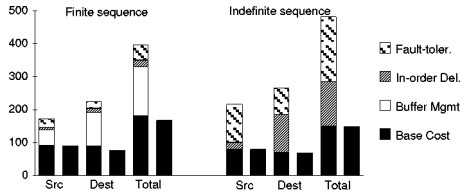


Figure 2. Active Messages on the CM-5: breakdown of overhead for 16-word messages

management (148 cycles), in-order delivery (21 cycles) and fault tolerance (47 cycles) (figure 2, left).

These results imply that the careful balance between functionalities offered and processing overhead is crucial in the design of high performance communication software. In gigabit/s networks the design constraints to deliver usable performance at small message sizes are even smaller. These lessons were crucial in the design of two generations of Fast Messages systems.

3. Fast Messages 1.x

The design of Fast Messages 1.0 for Myrinet provided an opportunity to apply the lessons of the networking community – low overhead to deliver performance to short messages, and a simple interface with the right guarantees to deliver performance to the application. By providing a few key services – buffer management, reliable and in-order delivery – the FM programming interface allows for a leaner, more efficient implementation of the higher level communication layers.

The first workstation cluster implementation of Fast Messages (FM) project [22] was built on Sparc 20 workstations interconnected with a Myrinet network. On this network FM achieves a short message latency of only 14 μs and a peak bandwidth of 17.6 MB/s, or 75% of the available 23 MB/s of I/O bandwidth when using programmed I/O. As a result of the design focus on short message performance, the value of $N_{1/2}$ is 54 bytes, with a bandwidth of 17.5 MB/s available for messages as small as 128 bytes.

3.1. Design of the Illinois Fast Messages 1.x

The FM 1.1 API consists of three functions `FM_send(.)`, `FM_send_4(.)`, and `FM_extract(.)` as shown in table 1. `FM_send(.)` and `FM_send4(.)` inject messages into the network. FM has an Active Message style interface that differs from a pure message passing paradigm by not having explicit receives. Instead, each message includes the name of a handler, which is a user-defined function that is invoked upon message arrival to process the carried data.

The `FM_extract()` primitive is used to service communication on the receive side, checking for incoming messages and executing the corresponding handlers. The user needs to call this primitive frequently to ensure the prompt processing of incoming communication in the host. However it does not need to be called for the network to make progress. FM provides buffering so that senders can make progress while their corresponding receivers are computing and not servicing the network.

Table 1
The primitives of the FM 1.1 API.

Function	Operation
FM_send_4(dest,handler,i0,i1,i2,i3)	Send a four word message
FM_send(dest,handler,buff,size)	Send a long message
FM_extract()	Process received messages

The FM interface is similar to the Active Messages model [30] from which it borrows the notion of message handlers. However, there are a number of key differences: the FM API offers stronger guarantees (in particular in-order delivery), a uniform handling of messages with respect to size, and there is not a request-reply scheme. Also, in contrast to Active Messages, where the send calls implicitly poll the network, FM's send calls do not normally process incoming messages, enabling a program to control when communications are processed.

In choosing which service guarantees to include during the design phase of FM, we gave careful consideration to the performance of the communications stack as a whole, not of FM as an isolated messaging layer. If a messaging layer's guarantees are too weak (i.e., they do not provide the functionality that applications expect), other messaging layers built on top will need to supply the missing functionality, incurring additional overhead in the process. On the other hand, if a messaging layer's guarantees are too strong (i.e., they provide more functionality than is generally needed), the messaging layer's common-case performance may be needlessly degraded. Analysis of the literature and our ongoing studies to support fine-grained parallel computing [5,14–16] have led to the conclusion that a low-level messaging layer should provide the following key guarantees:

- reliable delivery,
- in-order delivery, and
- control over scheduling of communication work (decoupling).

As mentioned in the previous section, studies of communication software costs [14] show that implementing guarantees like reliable and in-order delivery can increase communication overhead by over 200%. To reduce these costs careful consideration was given to exploiting hardware features. We found that by taking advantage of Myrinet features like very low error rate, absence of buffering in the network fabric, deterministic routing, link-level flow control by means of back-pressure, we only needed to add flow control and buffer management to provide reliable and in-order delivery. FM provides these, and its performance demonstrates that these guarantees need not be costly.

Figure 3(a) shows that the addition of buffer management and flow control is not substantially degrading performance. The different curves represent the performance

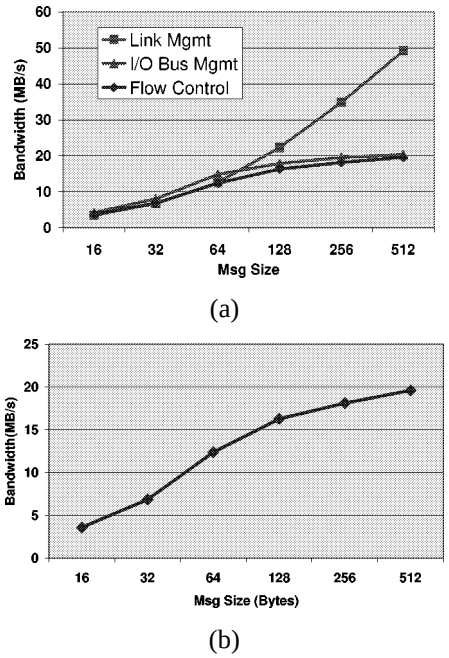


Figure 3. FM 1.x overhead: (a) overhead break-down; (b) overall performance.

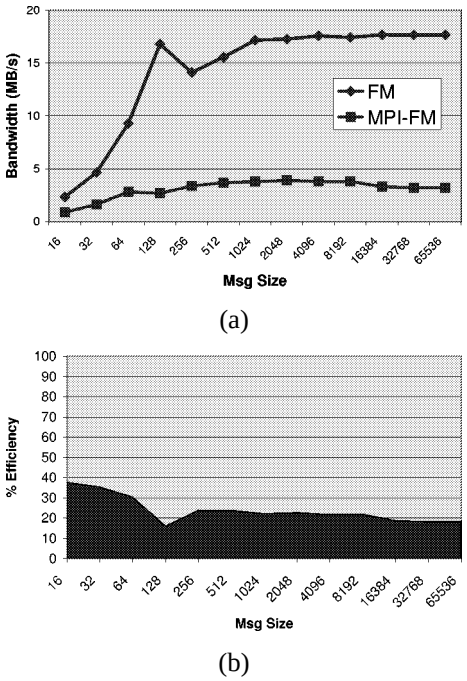


Figure 4. MPI-FM initial performance compared to FM: (a) absolute; (b) as a percentage of FM.

measured with the simplest code needed to operate the link DMAs, then with a few more lines to move data across the I/O bus, and finally with the flow management code added. The transport of data across the I/O bus is on the critical path and adds to the overhead, while flow control if properly designed can be overlapped with other operations. Similarly, the further addition of buffer management does not add substantial overhead, and leads to the final version of the FM code (figure 3(b)). A more detailed analysis of the FM 1.x design choices is reported in [22].

3.2. Evaluation of FM 1.x

The real measure of the effectiveness of a communication library is the level of performance that can be actually delivered to an application. Given the low-level nature of the FM interface, typical applications are language runtime supports or user level libraries. We selected MPI and BSD sockets as test applications, and experimented extensively with the former.

Figure 4 shows that the initial version of MPI-FM had poor performance, failing to deliver more than 35% of the underlying FM bandwidth. It was clear that the FM 1.x interface lacked several key features required for efficient layer composition. So the analysis of the MPI-FM inefficiencies turned into a study on how to design an API that makes it easy to deliver performance (see [19] for details).

The overhead originates from a number of memory-to-memory copies of the data taking place at the interface between MPI-FM and FM. The service guarantees we built into FM allowed a streamlined and thin implementation of the body of MPI-FM, for example making unnecessary the source buffering, timeout, and retry that would be otherwise required to provide reliable communication. But inefficiencies arose at the interface between layers, surprisingly for different reasons for each direction of transfer.

First, FM adopted its basic API from Active Message (AM) [30], and thus accepted (and presented) data as a single contiguous buffer. While sending, this approach charges the upper layers with the task of assembling/disassembling of messages. In many cases, this incurred an additional step (and copy) in performing common packet header operations for encapsulating, checksumming, etc.

Similarly to the send side, on the receive side the message is handed over to the handler as a single contiguous buffer. This required that the entire message had to be received into a staging buffer before the handler could start processing it and possibly copying it to the final destination. Such a scheme forced FM to perform an additional copy even when the availability of the destination buffer (i.e., preposted MPI receive) made it theoretically unnecessary.

Second, FM 1.x allowed the receiving process to decide when to service the network, however, it was unable to control the quantity of data presented at that time (all the pending packets were processed). In high speed networks, data can easily be transmitted far faster than a receiver can accept it. The presentation of the data before the application was prepared to accept induced additional layers of buffering and data copies.

In conclusion, the implementation of MPI-FM showed that the FM API was lacking flexibility in two crucial areas:

- presentation of data across layer boundaries,
- control over interlayer scheduling.

Addressing these shortcomings required some fundamental changes to the API, and motivated the design of a new version of FM.

4. Fast Messages 2.x

4.1. Design of Illinois Fast Messages 2.x

The FM 2.x API retains the service guarantees of FM 1.x, and adds support for gather-scatter, layer interleaving, and receiver flow control. The primary vehicle for these features is the addition of the *stream abstraction*, in which messages are viewed as byte streams and primitives are provided for the piecewise manipulation of data, both on the send and the receive side.

In the new FM 2.x interface (table 2) the old `FM_send(.)` primitive is replaced by `FM_send_piece(.)`, which can be called as many times as desired to send segments of a message of arbitrary size. Message boundaries are still honored (using the `FM_begin_message(.)` and `FM_end_message(.)` calls), but in the new API a message may be gathered from discontiguous regions of memory and marshalled into the network. Mirroring this abstraction on the receive side is the `FM_receive()` primitive, that can be called an arbitrary number of times from within a handler.

FM's *streaming interface* is an innovative way to deliver performance to higher-level messaging layers such as MPI. The idea is to enable a messaging layer to process a message as it arrives, without having to wait for the entire message to arrive before delivering it. Figure 5 shows the stages of the FM pipeline.[1] Note how the first part of the message is delivered before the last part of the message has even been passed to FM. As an added bonus, the streaming interface provides an elegant scatter/gather mechanism for handling noncontiguous data without extra memory copies.

`FM_receive()`, the function a handler uses to receive data from a message, copies the next set of bytes from the message to a given buffer. There are no limits on the number of bytes copied per invocation, as long as the total is no greater than the size of the original message. Specifically, the sequence of `FM_receive()`s need not match the sequence of `FM_send_piece()`s originally used to send the message. Furthermore, `FM_receive()` has blocking semantics; the next instruction in the handler will not execute until all the bytes specified are copied. This makes coding simple and natural.

In the first implementation of the streaming interface, which shipped with HPVM 1.0, lightweight threads were implemented with an ad hoc scheme that involved source code translation. The programmer was obligated to run his code through a "`streamify`" script that rewrote handlers so that they would explicitly save their state, switch control to a scheduler, and restore their state when control was switched back. While this scheme yielded good performance – the handlers and the scheduler did no more work than absolutely necessary – and was portable across operating systems and architectures, there were a number of drawbacks:

[1] "User buffer → FM (host)" and "host → NIC" run concurrently, but not in parallel. Hence, they are not drawn in a pipelined manner.

Table 2
The primitives of the FM 2.x API.

Function	Operation
FM_begin_message(dest, size, handler)	Start of a message to be sent
FM_send_piece(stream, buf, bytes)	Send a chunk of message
FM_end_message(stream)	End of a message to be sent
FM_receive(stream, buf, bytes)	Get a chunk of message
FM_extract(bytes)	Process received messages

- The `streamify` script was C-specific.
- Because `streamify` was not a complete C parser, but rather employed simple pattern-matching rules for rewriting, it imposed a number of limitations on handler coding style.
- Programmers had to help `streamify` identify handlers and variable declarations by including FM-specific `#pragma` directives at various places in their code.

In short, `streamify` was a big nuisance to programmers. For the latest generation of the streaming interface, we completely redesigned the implementation to eliminate the need for `streamify`. We benchmarked Win32 fibers (lightweight threads) and found that, while fiber creation time is high (26 μs on a 200 MHz Pentium Pro), fiber-switching time is extremely low (0.2 μs on the same platform). Hence, we made `FM_initialize()` preallocate a large number of fibers and give each fiber a pointer to the state that is shared between itself and the fiber scheduler (i.e., `FM_extract()`). The entry point of each fiber is a wrapper function that alternates between calling a handler and returning control to `FM_extract()`. The shared state is used by `FM_extract()` to point the wrapper function to the current handler and to the head of the receive queue. It is also used by the wrapper function to pass the handler's return code back to `FM_extract()`. `FM_receive()` is now an ordinary function (as opposed to a placeholder that gets rewritten by `streamify`), and there are no longer any coding style limitations on handlers. Furthermore, because fiber switching is so fast, the performance of the streaming interface is no worse than it was in the previous generation.

Thanks to the new design, the key problems identified in studies of FM 1.x are remedied as follows.

Gather/scatter. By performing a sequence of `FM_send_piece(.)` calls, the user can compose a message on the fly using any number of pieces, each of arbitrary size. Similarly, a receiver can employ a handler with a sequence of `FM_receive()` calls, allowing the efficient decomposition of a message into any number of pieces. Each call composes/extracts as many bytes as desired, and the number and sizes of the pieces need not match on the two sides. Examples include header attachment/removal in MPI-FM, and in protocol encapsulation in general (e.g., IP and TCP headers in TCP/IP hierarchy).

Layer interleaving. A second important benefit of the stream abstraction is the controlled interleaving of the FM

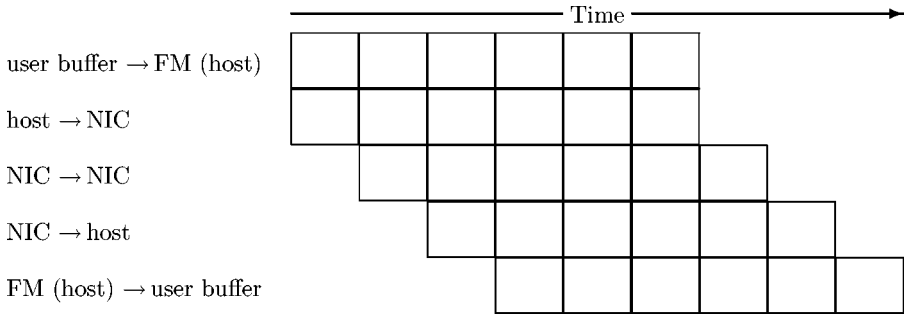


Figure 5. Pipelining within the FM layer (not drawn to scale).

and the application layers on the receive side. While everything runs within one user process, there is one thread for the application, in which the FM primitives are executed (including `FM_extract`), and one for each of the application-specific handlers. The typical message processing scenario within the handler is illustrated below:

```
int myHandler(FM_stream *str,
              unsigned sender)
{
  struct header myHeader;
  int msglen;

  /* get the header */
  FM_receive(&myHeader, str,
               sizeof(struct header));

  msglen = myHeader.length;

  if (myHeader.littlemsg)
    /* short message */
    FM_receive(littlebuf++, str,
               msglen);
  else
    /* long message */
    FM_receive(findBuf(msglen), str,
               msglen);

  return FM_CONTINUE;
}
```

The first `FM_receive()` call is used to extract just the message header (`FM_receive()` is executed within the application/FM thread). Then the handler reads the header fields, identifies the messages, and selects the buffer into which to copy the message payload (the handler is executed within its own thread). Finally, another call to `FM_receive()` with the selected buffer passed as second argument extracts the payload directly into the buffer (FM thread).

The interleaving makes possible the elimination of staging buffers for incoming messages. For example, in MPI-FM, using FM 1.x, we could not deliver an incoming message directly into its destination buffer, specified by the user through a pre-posted MPI receive call. The reason for this limitation was that incoming messages were handled by FM, while the buffer management occurred within MPI-FM, and the required exchange of information between the two layers (identity of message in one direction, pointer to the appropriate buffer in the other) was absent.

Receiver flow control. The FM 2.x interface also provides receiver flow control, allowing the receiver to control the rate at which data is processed from the network. This can eliminate network overruns of buffer pools, avoiding memory copies, and for some protocols, message discarding. In many applications, the ability to intentionally delay the extraction of the message until a buffer becomes available can simplify the buffer management. For example, receiver flow control enables zero-copy transfers in a significantly larger number of cases for both our Socket-FM and MPI-FM implementations.

Transparent handler multithreading. One of the differences between FM 1.x and FM 2.x is that handler execution is no longer delayed until the entire message has arrived, rather it is started as soon as the first packet is received. Since packets belonging to different messages can be received interleaved, the execution of several handlers can be pending at a given time. As it extracts each packet from the network, FM 2.x schedules the execution of the associated pending handler (figure 6). By having the interleaved packet reception transparently drive the handler execution, a number of benefits are achieved.

First, the handler multithreading combined with the stream abstraction allows arbitrary-sized data chunks to be composed/received, without any concern for packet boundaries. Second, handler multithreading plus packetization not only simplifies resource management, it can also increase performance by increasing effective pipelining in many cases. On a long message the handler can be processing one part of the message while the sender is still sending the rest. And the interleaving means that one arbitrarily long message from one sender does not block other senders.

The FM 2.x interface cleanly hides the physical packetization and handler multithreading by offering a clean sequential view of message reception. Except for the possibility of being descheduled on a `FM_receive()` call, a handler can be written as if the entire message had already

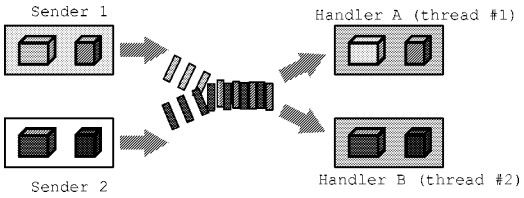


Figure 6. The interleving of packets drives the handler thread scheduling on the receiver.

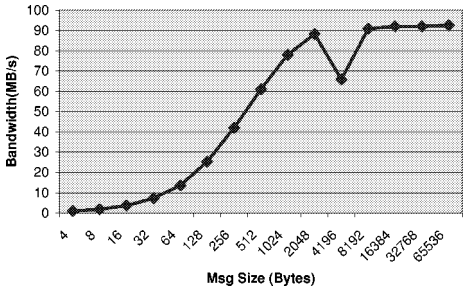
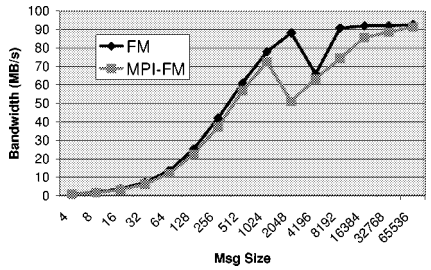
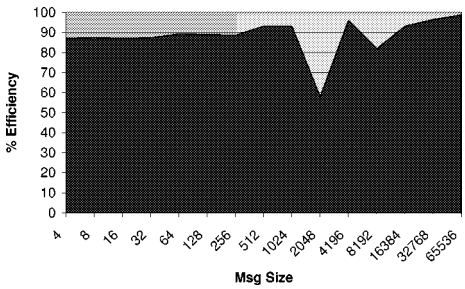


Figure 7. FM 2.1 performance on a 300 MHz Pentium II.



(a)



(b)

Figure 8. MPI-FM 2.0 performance compared to FM 2.0: (a) absolute; (b) as a percentage of FM.

been received. Moreover, the FM 2.x interface provides a logical thread for each message, avoiding explicit management of state sharing/isolation for complex messages.

4.2. Evaluation of FM 2.x

Figure 7 shows the performance achieved by FM 2.1 on a 300 MHz Pentium II: 9 μs minimum latency, 92 MB/s peak bandwidth, with $N_{1/2} \approx 256$ bytes. These values represent high absolute performance, compared to MPP interconnect performance and internal memory bandwidth. Similar to FM 1.x, a design attentive to short message performance shows in the $N_{1/2}$ values and in the rapid growth of the bandwidth curve.

The graphs of figure 8 show the improved efficiency of MPI-FM on top of FM 2.x, proving that the FM 2.x API can deliver a high percentage of its measured performance. MPI-FM achieves up to 98% of the FM bandwidth, with a minimum latency of 13 μs and a peak bandwidth of 91 MB/s. The key enhancements of FM 2.x (gather-scatter, layer interleaving, and receiver flow control) enable MPI on FM 2.x to eliminate many buffer copies, and avoid buffer pool overruns, delivering the underlying FM performance to the application. To further demonstrate FM 2.x's capabilities, we have implemented other APIs, including Shmem Put/Get and Global Arrays (both global address space interfaces). FM and the high-level user level interfaces are included in the High Performance Virtual Machine (HPVM), a complete suite of software tools for high performance computing on scalable clusters of PCs. The latest version of HPVM used for the reported measurements (HPVM 1.2) is available on our web site (`http://www-csag.ucsd.edu`).

5. Related work

Fast Messages is not the only approach to delivering high-performance communication by efficient protocol layering. Most related efforts involve either optimized implementations of heavyweight protocols, high-performance network hardware, or other high-performance low-level messaging layers. We now discuss projects in each of these categories.

High performance communication layers. Active Messages (AM) [30] has been one of the first realizations of high performance messaging layers. The AM project started as a communication library for the CM-5, and today some of its new implementations retain some of the features of the original version, like the specialized low-latency primitives for short transfers. A problem with specialized primitives is that they often fall short of the practical message size of overlying applications. For example, in the implementation of MPI-FM we found that the minimum length of the MPI header is 24 bytes (6 words), while low-latency primitives in Active Messages libraries are available for up to n words, where the value of n is 4, 5 or another integer depending on the specific AM release.

From the same group is Fast Sockets [27], an implementation of the Berkeley Sockets on top of Active Messages. One of the issues explored by Rodrigues at al. in their work is the elimination of unnecessary copies at the layer interface. The copy avoidance technique of *receive posting* in Fast Sockets is similar to what FM 2.x achieves with the layer interleaving, in which the user handler collaborates with FM to direct the incoming data directly into the destination buffer. The main difference is that the FM model

supports packetization and thus works with messages of arbitrary size.

The interplay between packetization and pipelining has been one of the main foci of the Trapeze project at Duke University. Trapeze [33] is a messaging layer for Myrinet networks developed for network memory and other distributed operating system services. It uses pipelining to reduce the latency of page-sized transfers across the network. The *cut-through delivery* technique of Trapeze adds a level of pipelining by allowing overlapped transfers of packet chunks, in addition to the traditional inter-packet overlapping. A number of design choices aimed at optimizing page transfers within the kernel (the use of DMA transfers on both sides, of interrupt-based notification) differentiate Trapeze from the FM user-level design.

Another high performance user-level messaging layer is U-Net [31]. Developed originally on a network ATM, it provides buffer management, demultiplexing in hardware but no flow control, and thus data can be lost due to overflow. Contrary to FM, U-Net and other messaging layers try to avoid the passage of data through kernel memory by performing a DMA transfer directly into the user buffer. The disadvantage of such a feature is that the user must declare in advance the regions of memory to be used for communication, so to allow the library to permanently pin them down.

In our experience such a scheme seems to lack the flexibility needed in building user-level libraries. In the case of MPI-FM, the buffers are provided by the MPI application and their location is not in general known in advance. A new version of U-Net called U-Net/MM [32] is under development which addresses this limitation by including a TLB on the network interface and coordinating its operation with the operating system's virtual memory subsystem. This mechanism would allow network buffer pages to be pinned and unpinned dynamically and thus messages can be transferred to and from any part of the application's address space.

A different kind of API is offered by the VMMC-2 [10]. Derived by the Shrimp project, VMMC-2 implements the notion of remote virtual memory mapping, in which a mapping is established between regions of memory of different machines. Once set up the mapping, stores into local pages of such regions are automatically propagated to the associated pages on remote nodes. There is little experience yet on the use of such abstraction to support layers of communication software.

In some respect similar to FM is the Real World Computing Partnership's PM [29]. Like FM, PM runs on clusters of Myrinet-connected workstations and performs flow control and buffer management. The main difference with FM is in the optimistic flow control mechanism, and variable-sized packets.

BIP [26] is another messaging layer developed for Myrinet at the Ecole Normale Superieure de Lyon. It has a more traditional message passing interface, with both blocking and non-blocking send/receive primitives, and offers unreliable and in-order delivery communication. It has been specifically designed to support standard message passing libraries like MPI and PVM, for which its interface represents a good match.

Optimized heavyweight protocols. One approach to fast communication that a number of researchers have taken is to start with traditional, heavyweight, kernel-mode protocol stacks and tune the implementations to deliver more performance. Frequently, these projects focus on the TCP and UDP stacks, but other protocols have been optimized as well. One of the largest performance penalties that occurs when sending large messages is memory copying, which occurs at each level in the protocol stack. (In TCP, the other big penalty is computing the TCP checksum, but this cost can be eliminated in some modern network interfaces by performing the checksum in hardware.) Hence, the most common optimization technique is to reduce the amount of data copying by sharing buffers across layers.

This is the approach taken by *fbufs* [9], which avoids data-touching overheads by remapping pages of data from one domain to another instead of copying. The Solaris operating system does something similar, but uses copy-on-write semantics to prevent wayward applications from corrupting data that are still "live" in the protocol stack [7]. *Container shipping* [23] and other protocol-stack optimizations [3] expand upon the basic fbufs technique. XTP [28] takes a different approach: It improves performance by providing high-level features such as multicast and priority control in a new, alternative heavyweight protocol.

The problem with all of these schemes, and one of the reasons that Fast Messages does not attempt a similar solution to the protocol layering performance problem, is that they perform poorly on small messages. And, for realistic message sizes – generally less than 256 bytes – memory copying is much less of a bottleneck than the various constant-time overheads [17]. Even the overhead to switch between user mode and kernel mode is too high for forthcoming networks. For perspective, note that on a gigabit network, about 1 KB of data can arrive in the time it takes just to switch modes.

6. Conclusions

We have described our experience with the implementation of user-level libraries on top of the FM library. The main finding is that overheads generated at the interface between layers can substantially reduce the communication performance seen by the applications. Our work exposes the need for a design of the low-level programming interface that specifically targets the efficient matching between layers, and identifies the crucial services required for such matching.

The gather/scatter functionality allowed a streamlined header attachment/removal process which previously required a copy of the transferred data. Layer interleaving

enabled the direct deposition of message data into the destination buffer rather than in a staging buffer, thus saving another copy. The redesigned API of our second generation communication layer, FM 2.0, adds these services in a flexible and performance-conscious way. Other benefits of the interface include receiver pacing, transparent handler multithreading, a simple and clean view of message reception, and a pipelining scheme that is the first to provide true end-to-end pipelining, from user buffer to user buffer.

The validity of our new design is shown by the peak bandwidth of a high-level library like MPI-FM that went from an initial 20% to the final 98% of the bandwidth made available by the FM layer. Such a result is even more relevant in view of the fourfold increase of absolute FM bandwidth from 17.6 MB/s to 92 MB/s, consequent to the migration form a Sparc to a x86 architecture.

Acknowledgements

The research described in this paper is supported in part by DARPA orders #E313 and #E524 through the US Air Force Rome Laboratory Contracts F30602-96-1-0286 and F30602-97-2-0121. Support from Microsoft, Intel Corporation, Hewlett-Packard, Myricom, Platform Computing, and Tandem Computers is also gratefully acknowledged. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809. Scott Pakin is supported by an Intel Foundation Graduate Fellowship. Mario Lauria has been supported in part by a NATO-CNR Advanced Science Fellowship.

References

[1] T.M. Anderson and R.S. Cornelius, High-performance switching with Fibre Channel, in: *Digest of Papers Compcon 1992* (IEEE Computer Society Press, Los Alamitos, CA, 1992) pp. 261–268.

[2] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic and W.-K. Su, Myrinet – a gigabit-per-second local-area network, IEEE Micro 15(1) (February 1995) 29–36. Available from `http://www.myri.com/research/publications/Hot.ps`.

[3] J.C. Brustoloni and P. Steenkiste, Effects of buffering semantics on I/O performance, in: *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, Washington (October 1996) pp. 277–291. Available from `http://www.cs.cmu.edu/afs/cs/user/jcb/papers/osdi96.ps`.

[4] CCITT, SG XVIII, Report R34, Draft Recommendation I.150: B-ISDN ATM functional characteristics (June 1990).

[5] A. Chien, J. Dolby, B. Ganguly, V. Karamcheti and X. Zhang, Supporting high level programming with high performance: The Illinois Concert system, in: *Proceedings of the 2nd International Workshop on High-level Parallel Programming Models and Supportive Environments* (April 1997) pp. 15–24.

[6] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini and J. Prusakova, High performance virtual machines (HPVM): Clusters with supercomputing APIs and performance, in: *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN (March 1997). Available from `http://www-csag.ucsd.edu/papers/hpvm-siam97.ps`.

[7] H.-K.J. Chu, Zero-copy TCP in Solaris, in: *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA (January 1996) pp. 253–264. Available from `http://playground.sun.com/~hkchu/zc-usenix.ps`.

[8] D.D. Clark, V. Jacobson, J. Romkey and H. Salwen, An analysis of TCP processing overhead, IEEE Communications Magazine 27(6) (June 1989) 23–29.

[9] P. Druschel and L.L. Peterson, Fbufs: A high-bandwidth cross-domain transfer facility, in: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC (December 1993) pp. 189–202. ACM SIGOPS, ACM Press. Available from `ftp://ftp.cs.arizona.edu/xkernel/Papers/fbuf.ps`.

[10] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis and K. Li, VMMC-2: efficient support for reliable, connection-oriented communication, in: *Proceedings of Hot Interconnects V*, IEEE (August 1997). Available from `http://www.cs.princeton.edu/shrimp/Papers/hotIC97VMMC2.ps`.

[11] Fiber-distributed data interface (FDDI) – Token ring media access control (MAC), American National Standard for Information Systems ANSI X3.139-1987, American National Standards Institute (July 1987).

[12] L.A. Giannini and A.A. Chien, A software architecture for global address space communication on clusters: Put/Get on Fast Messages, in: *Proceedings of High-Performance Distributed Computing Conference* (1998). Available from `http://www-csag.ucsd.edu/papers/hpdc7-giannini.ps`.

[13] R. Gusella, A measurement study of diskless workstation traffic on Ethernet, IEEE Transactions on Communications 38(9) (September 1990) 1557–1568.

[14] V. Karamcheti and A. Chien, Software overhead in messaging layers: Where does the time go? in: *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, Association for Computing Machinery (October 1994) pp. 51–60. Available from `http://www-csag.ucsd.edu/papers/asplos94.ps`.

[15] V. Karamcheti and A.A. Chien, A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D, in: *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, Santa Margherita Ligure, Italy (June 1995) pp. 298–307. Available from `http://www-csag.ucsd.edu/papers/cm5-t3d-messaging.ps`.

[16] V. Karamcheti, J. Plevyak and A.A. Chien, Runtime mechanisms for efficient dynamic multithreading, Journal of Parallel and Distributed Computing 37(1) (1996) 21–40. Available from `http://www-csag.ucsd.edu/papers/rtperf.ps`.

[17] J. Kay and J. Pasquale, The importance of non-data touching processing overheads in TCP/IP, in: *Proceedings of the ACM Communications Architectures and Protocols Conference (SIGCOMM)*, San Francisco, CA (September 1993) pp. 259–269. Available from `http://www-csl.ucsd.edu/CSL/pubs/conf/sigcomm93.ps`.

[18] J. Kay and J. Pasquale, Profiling and reducing processing overheads in TCP/IP, in: IEEE/ACM Transactions on Networking (December 1996). Available from `http://www-cse.ucsd.edu/users/pasquale/Papers/profTCP96.ps`.

[19] M. Lauria and A. Chien, MPI-FM: High performance MPI on workstation clusters, Journal of Parallel and Distributed Computing 40(1) (January 1997) 4–18. Available from `http://www-csag.ucsd.edu/papers/jpdc97-normal.ps`.

[20] M. Liu, J. Hsieh, D. Hu, J. Thomas and J. MacDonald, Distributed network computing over Local ATM Networks, in: *Supercomputing '94* (1995).

[21] S. Pakin, V. Karamcheti and A.A. Chien, Fast Messages: Efficient, portable communication for workstation clusters and MPPs, IEEE Concurrency 5(2) (April–June 1997) 60–73. Available from `http://www-csag.ucsd.edu/papers/fm-pdt.ps`.

[22] S. Pakin, M. Lauria and A. Chien, High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet, in:

Proceedings of the 1995 ACM/IEEE Supercomputing Conference, Vol. 2, San Diego, CA (December 1995) pp. 1528–1557. Available from `http://www-csag.ucsd.edu/papers/myrinet-fm-sc95.ps`.

[23] J. Pasquale, E.W. Anderson and K. Muller, Container Shipping: Operating system support for I/O-intensive applications, IEEE Computer 27(3) (March 1994) 84–93.

[24] J. Postel, User datagram protocol, RFC 768, Internet Engineering Task Force (August 1980). Available from `ftp://ds.internic.net/rfc/rfc768.txt`.

[25] J. Postel, Transmission control protocol, RFC 793, Internet Engineering Task Force (September 1981). Available from `ftp://ds.internic.net/rfc/rfc793.txt`.

[26] L. Prylli and B. Tourancheau, Protocol design for high performance networking: a Myrinet experience, Technical Report N. 97-22, LIP, Ecole Normale Superieure de Lyon (July 1997). Available from `http://www-bip.univ-lyon1.fr/`.

[27] S. Rodrigues, T. Anderson and D. Culler, High-performance local-area communication using Fast Socket, in: *Proceedings of the USENIX 1997 Technical Conference*, San Diego, CA (USENIX Association, January 1997). Available from `http://now.cs.berkeley.edu/Papers2/`.

[28] W.T. Strayer, B.J. Dempsey and A.C. Weaver, *XTP: The XPress Tranfer Protocol* (Addison-Wesley, Reading, MA, 1992).

[29] H. Tezuka, A. Hori and Y. Ishikawa, PM: A high-performance communication library for multi-user parallel environments, Technical Report TR-96-015, Tsukuba Research Center, Real World Computing Partnership (November 1996). Available from `http://www.rwcp.or.jp/papers/1996/mpsoft/tr96015.ps.gz`.

[30] T. von Eicken, D. Culler, S. Goldstein and K. Schauser, Active Messages: a mechanism for integrated communication and computation, in: *Proceedings of the International Symposium on Computer Architecture* (1992) pp. 256–266.

[31] T. von Eicken, A. Basu, V. Buch and W. Vogels, U-Net: A user-level network interface for parallel and distributed computing, in: *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (December 1995) pp. 40–53. Available from `http://www2.cs.cornell.edu/U-Net/papers/sosp.pdf`.

[32] M. Welsh, A. Basu and T. von Eicken, Incorporating memory management into user-level network interfaces, in: *Hot Interconnects V*, Stanford, CA (August 1997). Available from `http://www.cs.cornell.edu/U-Net/papers/hoti97.ps`.

[33] K.G. Yocum, J.S. Chase, A.J. Gallatin and A.R. Lebeck, Cut-through delivery in Trapeze: an exercise in low-latency messaging, in: *HPDC-6*, Portland, OR (August 1997).

Mario Lauria is currently a postdoctoral researcher in the Department of Computer Science and Engineering at the University of California, San Diego. Mario Lauria received his Laurea degree in EE and his Ph.D. in EE&CS from the Federico II University of Naples, Italy, in 1992 and in 1997, respectively, and an M.S. in CS from the University of Illinois at Urbana-Champaign in 1996. After a year as a lecturer at the University of Naples and a year as a postdoc at the University of Illinois on a NATO-CNR Science Fellowship, he is continuing at the UCSD his research activity in the Concurrent Systems Architecture Group, of which he is a member since 1994. His research interests include network architectures, hardware/software support for high speed communications in clusters, cluster architectures for HPC.

E-mail: mlauria@csag.ucsd.edu

Scott Pakin is in the doctoral program in computer science at the University of Illinois at Urbana-Champaign. He is also a member of Andrew Chien's Concurrent Systems Architecture Group. His research interests include high-speed communication architectures for workstation clusters and coordinated scheduling. He received his MS in computer science from the University of Illinois at Urbana-Champaign in 1995 and his BS in mathematics/computer science from Carnegie Mellon University in 1992.

E-mail: pakin@cs.uiuc.edu

Andrew A. Chien is the Science Applications International Corporation Chair Professor in the Department of Computer Science and Engineering at the University of California, San Diego, and is affiliated with the National Computational Science Alliance (NCSA) and the National Partnership for Advanced Computational Infrastructure (NPACI). From 1990 to 1998, Andrew was a faculty member at the University of Illinois, Department of Computer Science, and remains an adjunct faculty member there. Andrew's research involves networks, network interfaces, and the interaction of communication and computation in high performance systems. His work also involves compilation techniques for high performance object systems. Professor Chien received his B.S. degree in EE from the Massachusetts Institute of Technology in 1984. He also received his M.S. and Ph.D. degrees in computer science from M.I.T. in 1987 and 1990, respectively. He is the author of over seventy research papers and book chapters on networks, architecture, compilers and programming languages. He has served on numerous program committees as well as program chairman for the 1999 ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming, Program Vice Chair of the 1995 IEEE International Parallel Processing Symposium, and as an Associate Editor for IEEE Transactions on Parallel and Distributed Systems. Professor Chien was awarded the National Science Foundation Young Investigator Award in 1994. In 1995, he received the C.W. Gear Outstanding Faculty Award and, in 1996, he received the Senior Xerox Award for Outstanding Research.

E-mail: achien@cs.ucsd.edu