Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors

Scott Pakin Vijay Karamcheti Andrew A. Chien

Department of Computer Science
University of Illinois
1304 W. Springfield Avenue
Urbana, IL 61801
{*pakin, vijayk, achien*}*@cs.uiuc.edu*

January 27, 1997

Abstract

Illinois Fast Messages (FM) is a low-level software messaging layer designed to meet the demands of high performance network hardware. It delivers much of the hardware's raw performance to both applications and higher-level messaging layers. FM presents an architectural interface which is both portable and amenable to high-performance implementations on massively-parallel computers and networks of workstations. By providing key services—buffer management and ordered, reliable delivery—FM enables the simplification and streamlining of higher level communication layers. FM also decouples the processor and the network, lending control over scheduling communication processing to software built on top of FM. This, in turn, minimizes communication's impact on local computation performance (e.g. by preserving cached data).

We have built several implementations of the FM interface on the Cray T3D and a Myrinet-based workstation cluster. These implementations demonstrate that FM can deliver much of the underlying hardware's performance. On the Cray T3D, FM achieves 4.9 μs total overhead and 6.1 μs latency for a minimal-sized message, substantially better than other messaging layers on that platform, most notably PVM. T3D FM reaches a peak bandwidth of 112 MB/s out of a hardware limit of 130 MB/s. On a SPARCstation-based subset of our Myrinet cluster, FM achieves 4.4 μs overhead and 13.1 μs latency for a minimal-sized message, again substantially better than other messaging layers on that platform, most notably the Myrinet API. Its peak bandwidth of 17.5 MB/s is fairly close to the SBus programmed I/O limitation. FM's performance continues to improve with newer networking hardware and advances in our implementation techniques. On a pair of Pentium Pro-based PCs, Myrinet FM reaches a peak bandwidth of 56.3 MB/s with a minimum latency of 11.5 μs.

Keywords: communication, message-passing, massively-parallel processing, workstation clusters, network interface, software messaging layer, flow control, reliable delivery, in-order delivery

1 Introduction

Since the earliest days of the ARPAnet, users have acknowledged the importance of networking to high-performance as well as everyday computing. However, with the rapid growth of the Internet and increasing recognition of the social, educational, and commmercial importance of the National Information Infrastructure (NII), the importance of networking has recently become even more prominent. The utility of a computer is increasingly a function of the data which it can access and how fast it can access the data. National networks which once consisted of 56 kilobit/second links now contain links with hundreds of megabits/second throughput, and will soon include links with hundreds of gigabits/second throughput. Building software and hardware which enables computers to support such high network data rates is a significant challenge.

Networking software has evolved along with the ARPAnet and Internet, but subject to a widening range of needs: universal interoperabilty, reliability, extensible networks, and higher performance. However, in most cases, the available network bandwidth has not been extremely high, and as a result, much networking software achieves only a fraction of even Ethernet's 10 Mbps[1] performance for typical message size distributions. Recent efforts driven by academia, industry, and the various gigabit testbeds have pushed towards delivering gigabit performance to the application through standard interfaces and protocols such as TCP/IP. While such efforts have made significant progress [7, 10], high data rates are achieved only for extremely large messages, as exemplified by the CASA gigabit testbed [13]. In practice, however, typical TCP/IP packet size distributions are almost exclusively under 200 bytes of data [24], with a large fraction under 10 bytes [9]. The ability to sustain high data rates on messages of those sizes remains an imposing challenge.

An important concurrent trend to the rapid increase in network hardware performance is the rapid increase in microprocessors' computing performance. Low-cost microprocessors are arguably as powerful today as any computer processors that can be built. The unavoidable implication of low-cost microprocessors with high absolute performance is that high-performance systems must be constructed from scalable ensembles of microprocessors. Indeed the national High Performance Computing and Communications Initiative's focus on scalability and large-scale parallel systems (hardware and software) clearly reflects this reality. Virtually all high-performance computing vendors now market systems based on ensembles of microprocessors. However, the use of parallel ensembles to achieve high performance places a premium on efficient coordination and data movement. While hardware networks can achieve latencies of less than one microsecond, the software overhead for communication is often several orders of magnitude larger. New software architectures are needed to deliver low latency communication which is essential for efficient coordination and data movement in a parallel ensemble.

Prognostications of the future structure of the NII typically include high-performance servers of information, computation, and other specialized services embedded in a high speed network fabric with hundreds of millions of other hosts. There are two main system architectures which are likely candidates for these servers: massively-parallel processors (MPPs) and networks of workstations (NOWs). Both are attractive because of their ability to scale. However, both are critically dependent on internal communication performance to be effective servers, and any NII server is dependent on excellent external networking in order to support NII service. Interestingly, in recent years, MPP and NOW hardware has become increasingly similar, as both are driven by the significant cost advantages of high volume products. Hence, many of the issues involved in delivering communication performance within such systems have converged as well.

[1] We use "Mbps" to refer to millions (10^6) of bits per second and "Mb/s" to refer to megabits (2^{20}) per second. Medium rates are traditionally listed in Mbps.

The goal of the Illinois Fast Messages project is to exploit this convergence to develop communications technology that spans both MPPs and NOWs, supporting both intra-cluster communication and high speed external networking. In this paper, we describe the initial design and early progress on Illinois Fast Messages (FM), a portable, high-performance, low-level messaging interface designed to meet the challenges of high speed networking. FM implementations not only deliver high performance, but also provide three key guarantees that enable streamlined implementations of higher level protocols atop FM. These guarantees are:

- Reliable delivery,
- In-order delivery, and
- Decoupling of processor and network.

While many current-day high speed networks have extremely low channel error rates, providing reliable delivery also requires buffer management and flow control. FM not only provides these, but its performance demonstrates that these guarantees need not be costly. The small cost these guarantees do incur is offset by the additional performance gains they offer to higher protocol levels in the form of more simplified, streamlined control. FM provides in-order delivery, which saves higher protocols from the burden of storing and reordering messages. Finally, FM provides buffering which decouples the processor and network. This allows communication to be truly one-sided. That is, a computation can control when it processes communications and the network can make progress in the face of this deferral. While FM is certainly not the only approach to delivering high-performance communication (See Sidebar B or [44]), FM's guarantees define a distinct design point of demonstrated utility.

FM implementations are currently available for the Cray T3D MPP and Myrinet-based workstation clusters. Both implementations achieve high performance. On the Cray T3D, FM achieves a minimum of $4.9\,\mu$s total (send + receive) overhead and $6.1\,\mu$s latency, substantially better than other messaging layers such as Cray's own PVM implementation. On a Myrinet-connected workstation cluster of SPARCstations, FM achieves $4.4\,\mu$s total overhead and $13.1\,\mu$s latency, also better than the vendor-supplied messaging layer. The performance continues to improve with newer networking hardware and advances in our implementation techniques. See Sidebar C for our latest performance results.

Using two widely-accepted standard interfaces, UNIX sockets [33] and the Message Passing Interface (MPI) [35], we demonstrate that FM can be used to build higher level protocols which deliver much of the underlying hardware performance. The key design choices and their ramifications are discussed in detail. In the remainder of the article, we motivate and describe the design of FM, detailing its performance on multiple platforms, and advantages over alternative messaging layers. In Section 2, we delve into more detail about messaging layer design and put the various issues in context. In Section 3, we introduce FM and explain the goals of the FM project. The Cray and workstation versions of FM are described in Sections 4 and 5, respectively. We present some current applications of FM in Section 6. In Sections 7 and 8, we discuss the implications of our work and some key related work. Section 9 recaps the paper and Section 10 points out several future research directions.

2 Background

One can observe a number of trends in high-performance local area networks (LANs):

- Improved link data rates,
- Increased reliability,
- Switch-based interconnects, and
- "Smart" network interfaces and switches.

Today, most local area networks are interconnected via Ethernet [36]. Ethernet is a bus interconnect that runs at 10 Mbps in almost all installations. However, 10 Mbps is far too little bandwidth for new, network-intensive applications, such as those utilizing multimedia. For example, even television-sized video displayed with 8 bits/pixel and running at 30 frames/second requires approximately 63 Mbps. Compression (e.g. MPEG or MPEG-II) can greatly reduce the bandwidth requirement at some cost in latency. However, higher resolutions, higher frame rates, and less efficient, but faster, encodings are often of interest. Fortunately, a number of new, higher-bandwidth networks have recently hit the market. These include FDDI [18], 100 Base-T Ethernet [22], FibreChannel [46], Myrinet [4], and ATM/SONET [6] and currently run from 100 Mbps to over 600 Mbps, with the ability to scale to Gbps bandwidths and beyond.

Not only are these new networks faster, but they are often more reliable. That is, dropped or corrupted data packets occur less frequently. This is primarily due to buffering within the network and different media access control strategies. In Ethernet, for example, nodes can write data onto the bus without first ensuring that no other node is simultaneously doing the same. Data collisions lead to dropped or corrupted packets. In contrast, on non-bus-based networks, data is buffered until it is safe to transmit it over the next link.

Already, arbitrarily-interconnected switches have surpassed buses as the topology of choice for high-speed networks. Switches have a number of advantages over buses. Aggregate bandwidth can be increased by adding additional switches, while bus bandwidth is fixed. Switches also allow multiple sets of nodes to communicate in parallel, while buses serialize communication.

Regardless of the type of network, modern network interfaces are less and less frequently implemented as passive devices. Instead, the network interface boards often contain one or more of the following:

- Finite state machines, which generally include a DMA engine so the microprocessor can move data to or from host memory without host CPU involvement,
- Protocol-specific hardware, for example, support for ATM AAL5 segmentation and reassembly, and
- Programmable microprocessors and a modest amount of memory (typically a few hundred kilobytes up to a megabyte).

The importance of "intelligent" interfaces is that they enable more concurrency in protocol processing. That is, initial protocol processing can proceeed while the CPU is busy computing. Also, these interfaces may be faster at their dedicated tasks than a general-purpose CPU, in part because they are always servicing the network, not dividing their time between communication and computation.

There is every reason to believe these trends towards fast, reliable, switch-based interconnects and powerful network interfaces will continue, because they help deliver high-speed communication—a necessity for modern computer usage. It is therefore important to design communication software under the assumption that the underlying hardware will deliver high bandwidth (in the Gbps range), be composed of interconnected switches (often implying an absence of broadcast capabilities), and contain special logic to enable some of the protocol processing to be offloaded from the host CPU.

Unfortunately, software for these new networks generally lags well behind the hardware in terms of performance. Legacy software structures are still prevalent. The problem with this existing networking code is the built-in assumptions that are no longer valid. Primarily, software assumes that the network is so slow relative to the host CPU and memory system that it can afford to use poorly-optimized, general-purpose data structures and control paths. For example, operating systems are heavily involved in communication and tend to make copies of message data at multiple stages between the network hardware and the user's application's buffers. Both of these add substantial overhead to message processing, overhead that can be tolerated on slow networks, but not on more recent, faster networks. To put this overhead in perspective, note that on a 1 Gbps network, software has only $\approx 8\,\mu$s to process a 1 KB packet—a brief amount of time for protocol processing, even on a fast computer.

To drive down overhead, high-performance communication software removes the operating system from the critical path of communication. On gigabit networks, there is not enough time to switch into supervisor mode, load and transfer control to a device driver, process the message data, reload the user's application, and switch back to user mode for every packet sent or received. Hence, the operating system is used solely for initializing the network interface, establishing connections with other nodes, mapping network interface control registers and memory buffers into user space, and tearing down connections when communication is complete. All data transmission is performed at user-level. This approach is exemplified by Hamlyn [8], Cranium [34], U-Net [43], and SHRIMP [3]. Since operating systems are traditionally used for process protection, these systems all exploit the system's virtual memory hardware for protection, a much lower-overhead mechanism than system calls.

3 Illinois Fast Messages

Illinois Fast Messages (FM) is a low-level messaging layer which delivers high performance with the key guarantees that higher-level messaging layers need for efficiency. Messaging is a well-established communication mechanism of wide utility for parallel coordination [19]. The FM interface design is similar to Berkeley Active Messages [44] on the CM-5—a model of simplicity and functionality—but with a few critical distinctions detailed in this section. Primarily, FM borrows the notion of message handlers and uses essentially the same API. However, FM expands upon Active Messages by imposing stronger guarantees on the underlying communication.

The FM interface is platform-independent and therefore easily adaptable to new architectures. We have implemented FM on two platforms: the Cray T3D and workstation clusters connected by a Myrinet network. Independent from us, a group at Fujitsu implemented FM on FibreChannel [27], which further substantiates the portability of the interface. The base philosophy of FM is described in this section, and the various implementations are detailed in subsequent sections. For each implementation, we explain the flow control and buffering schemes we used to implement sender-receiver decoupling. Finally, we present performance data to demonstrate that even though FM has a rich set of features, it performs comparably with less robust messaging layers.

3.1 FM 1.1 Interface

The FM 1.1 interface consists of three functions (Table 1). `FM_send()` and `FM_send_4()` inject messages into the network. (The latter call is intended to send data from registers and thereby eliminate the need for memory traffic). As with Active Messages, each message has a corresponding handler function, indicated in its header, which is executed to process the message data. The `FM_extract()` call services the network, dequeueing pending messages and executing their corresponding handlers. Handlers bear the responsibility for moving the message data (if necessary) from temporary FM buffers into user memory. The FM interface provides a simple buffer management protocol: once an FM send function returns, FM guarantees that the buffer containing the message can be safely reused.

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

Table 1: FM 1.1 API

One important property of `FM_extract()` is that it need not 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. Also, in contrast to Active Messages, where the send calls implicitly poll the network, FM's send calls do not normally process incoming messages,[2] enabling a program to control when communications are processed.

3.2 Guarantees

The most important facet of messaging layer design is the choice of service guarantees to provide to higher-level messaging layers and applications. 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 [12, 28, 29, 30] have led to the conclusion that a low-level messaging layer should provide the following key guarantees:

- Reliable delivery,
- Ordered delivery, and
- Control over scheduling of communication work (decoupling).

Previous studies of communication cost in the CM-5 multicomputer system [28] indicate that software overhead for reliability, fault tolerance, and ordering can increase communication cost by over 200%, yet many low-level communication systems simplify their implementation by discarding packets. When networks were unreliable, this practice made sense, but modern networks are highly reliable, so such discarding is the major source of data—and therefore, performance—loss. Experience with messaging layers in multicomputers [39], shared memory systems [11], and high speed wide-area networks indicate that cache interference is a critical effect for both communication

[2] `FM_send()` and `FM_send_4()` call `FM_extract()` only when necessary to avoid buffer deadlock.

and local computational performance. Providing control over the scheduling of communication work allows programs to control their cache performance, in many cases enabling more efficient computation and communication. However, allowing such scheduling control requires that the processor be decoupled from the network, so that senders are not blocked waiting for receivers to extract messages from the network.

3.3 Extensibility

The FM interface was designed to support the building of higher level messaging layers. In particular, the guarantees provided by FM are specifically designed to enable streamlining of higher level protocols. For example, FM provides reliable delivery, so protocols may be able to eliminate retransmission techniques to deal with lost packets. FM also provides ordering, so higher level protocols can be simplified by eliminating reordering code. To demonstrate these benefits, in Section 6, we will discuss how they were used in the context of FM 1.1 to design several higher level communication layers and the lessons learned from that experience that are driving enhancements of the FM 2.0 interface (Sidebar C).

While FM provides significant guarantees to higher level messaging layers, the interface is also designed to allow easy extension of functionality. For example, although FM provides reliable delivery—ensuring delivery in all cases except uncorrectable hardware communication errors—some applications might require absolute data integrity. The use of memory buffers at the FM interface allows easy addition of buffer redundancy schemes. Likewise, applications requiring data security can easily layer encryption atop FM by writing a routine which encrypts data in place when passed a pointer to a memory buffer.

4 Cray T3D FM Implementation

Our first implementation of FM was on the Cray T3D [14]. This implementation exploits the high-speed interconnect and featureful network interface unit to achieve excellent communication performance. The T3D supports up to 4096 processors and is a shared address space machine with no hardware support for cache coherence. However, because memory access is non-uniform both in mechanism (special addressing mechanisms for remote locations) and performance (4–5 times slower than accesses to remote memory [1]), a message-passing programming style can be beneficial for performance as well as portability and relative ease of programming. The key relevant features of the T3D are all implemented by the network interface unit (Figure 1), called the annex or DTB annex. The annex supports remote reads and writes as well as atomic fetch & increment and atomic swap operations that can be accessed by remote processors. Because the network already guarantees reliable, in-order delivery, the most important aspects of implementing the FM interface on the T3D are decoupling communication and computation and achieving high performance. The multiplicity of mechanisms in the annex enables distinct implementations of the FM interface, each optimized for a different model of usage. Note that with a uniform interface, an application program can easily change back and forth between implementations as the performance and communication patterns dictate. We have constructed two implementations of our interface on the T3D: Push FM and Pull FM.

4.1 Push FM on the T3D

Push FM is a traditional style of messaging implementation which pushes data from the source to the destination, and buffers it for the receiving processor. With push messaging, FM eagerly

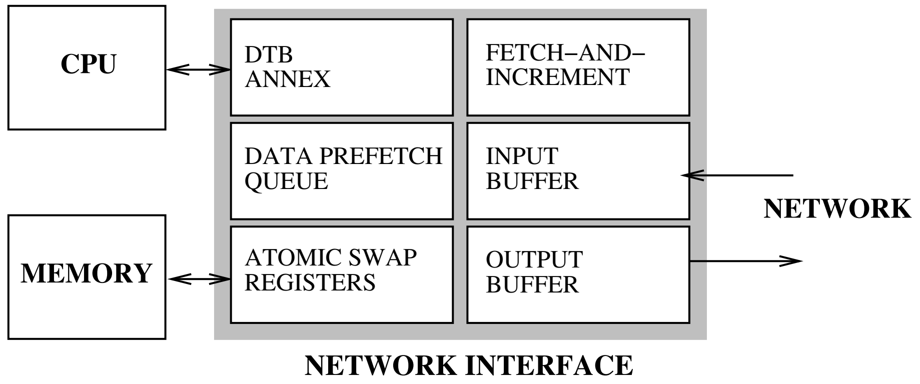


Figure 1: Cray T3D machine model

propagates messages from the sender to the receiving node's memory. As Figure 2 illustrates, in Push FM, the sender uses the fetch & increment operation against the receiver's fetch & increment register to acquire a unique index into a preallocated message buffer at the receiving node (Step 1). With this index, the sender moves the message data into that buffer, using remote write operations (Step 2). Because the fetch & increment operation atomically handles the buffer allocation among competing senders, there is no chance that another node will try to use the buffer simultaneously. When the receiver wishes to process the message, it simply reads the message from the buffer in its local memory (Step 3).

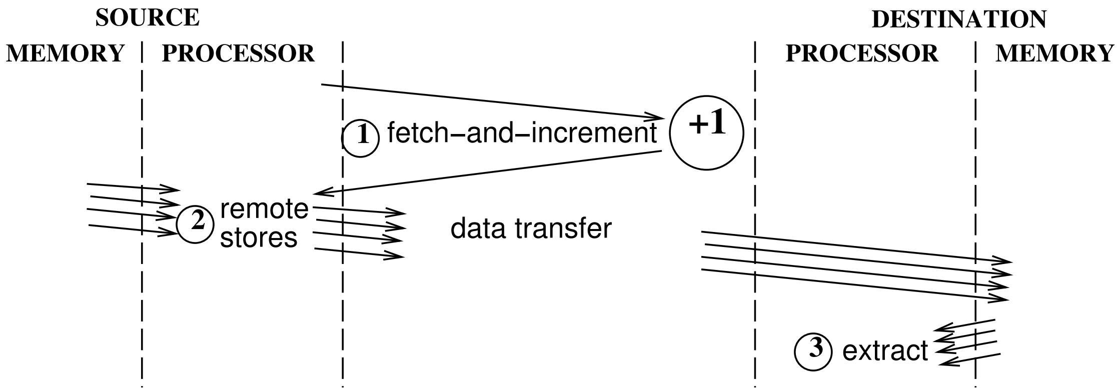


Figure 2: Push messaging

While push messaging minimizes latency at low network loads, performance can degrade if there is output contention or if the receiver allows its incoming buffers to fill by not servicing the network often enough [29]. If messages arrive at a receiver faster than the receiver's memory can process them or the receiver extracts them, the writes back up into the network, adding to network contention and increasing average latency. This effect has been observed in many irregular parallel computations, and provides the impetus for Pull FM.

4.2 Pull FM on the T3D

To avoid output contention, Pull FM moves data lazily—only when the receiver is ready to process it. Pull messaging is enabled by the T3D's shared address space, which allows receivers to reach into the sender's memory and pull the message data into local memory. Pull messaging effectively eliminates output contention and is therefore particularly important for irregular computations.

Pull FM (Figure 3) first copies the message data into a local memory buffer (Step 1) and then uses atomic swap to link the message into the receiver's receive queue (Steps 2 and 3). This queue is a distributed linked list whose head and tail are stored at the receiver. Note that data remains in the sender's local memory. When the receiver wants to read a message, it pulls the message into its local memory using remote memory read operations (Step 4). The latency of these remote reads is masked by using the T3D's prefetch hardware. Finally, the receiver deallocates the message buffer by storing a "reclaim" flag into the buffer (Step 5).

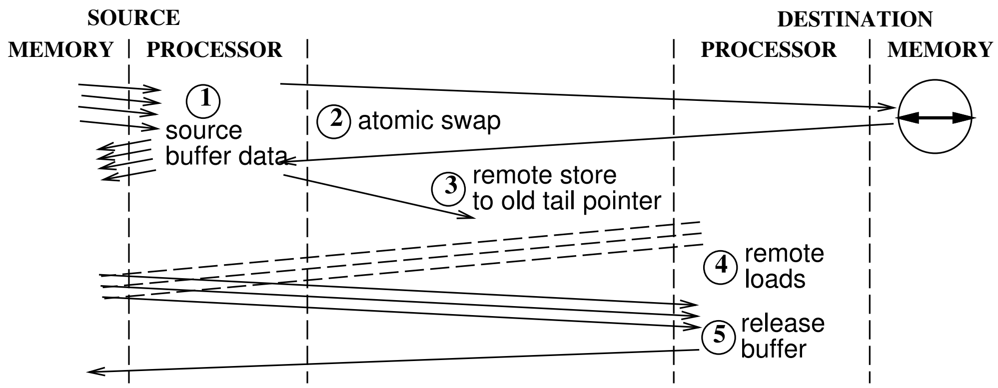


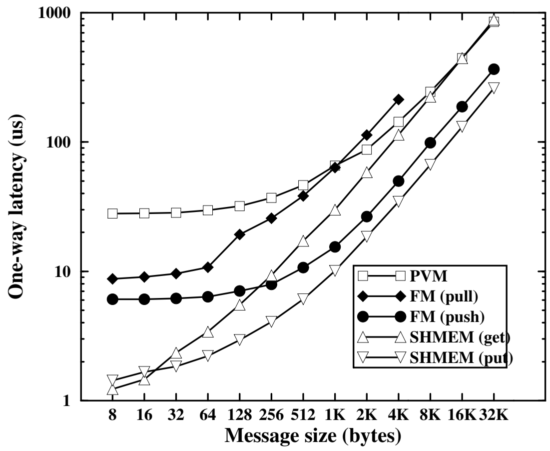
Figure 3: Pull messaging

Pull FM ensures that nodes are never swamped with data because each receiver pulls data across at its own pace. This effectively eliminates output contention, because the data arrival rate is matched to the pulling rate of a receiver. Furthermore, enqueuing messages is a comparatively low-overhead operation and is therefore beneficial for fine-grained applications. Note that pull messaging also serendipitously eliminates the message buffer overflow problem. The buffer allocation problem is transferred back to the senders, where flow control of ill-behaved senders is easily achieved. One drawback of Pull FM is that extra work is required to pull data across the network, increasing the overhead and latency slightly compared to Push FM [29].

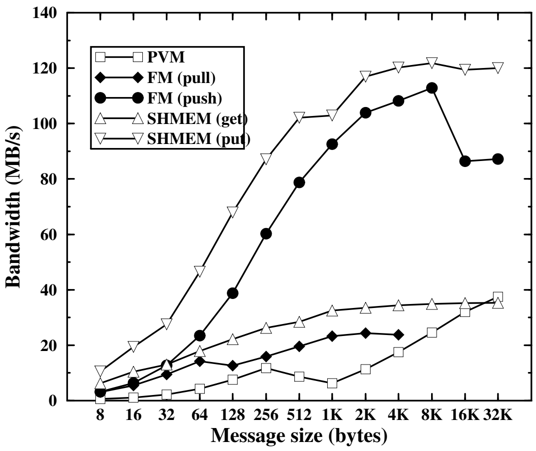
4.3 Performance

To place the performance of T3D FM in context, we compare it to two other vendor-supplied communication layers on the T3D: SHMEM [2] and PVM [20]. SHMEM is a low-level data movement—not messaging—library that copies data between addresses in the shared address space. PVM is a full-featured messaging layer that does both the end-to-end synchronization and buffer management required for traditional messaging. The PVM implementation was optimized by Cray for use on the T3D.

Performance comparisons are shown in Figure 4. Figure 4(a) compares FM's one-way latency to that of PVM and SHMEM (both with PUT and GET semantics). Figure 4(b) does the same for bandwidth. First, we focus on Push FM's latency. The Push FM latency curve is essentially level until around 256 bytes, after which it increases linearly. This is because the fixed overheads—especially the fetch & increment—dominate the latency until that point; afterwards the per-byte costs dominate. Push FM's latency is 6.1 μs for an 8-byte message and 366.8 μs for a 32 KB message. These times are close to the best achievable on the T3D hardware, as we will elaborate on shortly.



(a) Latency



(b) Bandwidth

Figure 4: FM Performance (T3D)

Pull FM's latency is comparable to Push FM's for small messages. However, for larger messages, Push FM is quite a bit slower due to the limited performance of the prefetch hardware. The jump in Pull FM's latency from 64–128 bytes stems from FM's use of 112-byte packet buffers. For messages larger than 112 bytes, segmentation and reassembly is required, adding overhead and reducing bandwidth. Still, for the types of messages Pull FM was designed to handle (i.e. the small messages used by fine-grained programs), Pull FM's 8.7 μs 8-byte latency is good, even if its 213.2 μs 4 KB latency[3] is a bit high. To compare these numbers to a commercial implementation of a widely-used messaging layer, we now turn our attention to the PVM curve. PVM's latency parallels Push FM's. However, PVM's latency is much higher than Push FM's for large messages and both FMs for small messages. Its latency ranges from 28.0 μs for 8-byte messages to 850.2 μs for 32 KB messages. Hence, even though FM also provides a messaging interface that supplies buffer management and flow control, its no-frills operation delivers superior performance to PVM's.

Finally, to put PVM and FM's latency in perspective, we compare it to that of SHMEM, which is little more than a set of functions for directly accessing the raw hardware and therefore extremely efficient. The two main mechanisms the SHMEM layer provides are PUT and GET. Because SHMEM PUT performs neither flow control, buffer management, nor notification of message arrival, its latency is fairly good: 1.4 μs for an 8-byte message, increasingly almost linearly to only 260.9 μs for a 32 KB message. Note however, that Push FM's latency approaches SHMEM PUT's as message size increases. Since Push FM and SHMEM PUT both rely on remote stores for data movement, this behavior is expected. SHMEM GET, however, uses a different mechanism—remote loads. Since remote stores are non-blocking, while remote loads block, Push FM's latency performance exceeds SHMEM GET's for messages larger than 128 bytes, after which Push FM's fixed overhead is sufficiently amortized. Still, SHMEM GET is capable of retrieving an 8-byte message in a

[3] Messages in Pull FM are limited to 4 KB.

rapid 1.2μs, although its 32KB latency, 885.9μs, exceeds even PVM's. What Figure 4(a) shows, therefore, is that even though FM performs flow control and buffer management like PVM, its latency is comparable to that of the lower-level SHMEM layer.

Figure 4(b) demonstrates FM's bandwidth over a range of message sizes. In general, the performance follows the same ordering as in Figure 4(a). However, there are fewer crossovers than in the latency graph. As before, we look first at Push FM. Push FM's bandwidth increases smoothly until the message size exceeds 8KB, after which point it drops slightly. That drop in performance is attributable to cache effects—the Alpha has only 8KB of data cache, and there is no additional off-chip cache on the T3D. Even so, Push FM's 112.9MB/s for 8KB messages is quite good and reflects the limitation of the Alpha's write buffer, which peaks at $\approx$ 130MB/s. Pull FM's bandwidth is comparable to Push FM's bandwidth for small messages, but is lower for large messages. The lower large-message bandwidth is attributable primarily to the peak bandwidth of the prefetch queue, approximately 32MB/s. As in the latency curve, Pull FM's bandwidth curve exhibits a drop in performance from 64–128 bytes due to the segmentation and reassembly used for messages larger than 112 bytes. Pull FM's bandwidth ranges from 3.1MB/s for 8-byte messages to 23.8MB/s for 4KB messages.

We now compare the bandwidth of FM versus PVM. Both versions of FM have much better bandwidth than PVM for small messages. For 8-byte messages, PVM's bandwidth—0.6 MB/s—is one-fifth of FM's. For larger messages, Push FM's bandwidth is still over twice PVM's. However, because PVM is not bandwidth-limited by the prefetch queue, its bandwidth eventually surpasses that which Pull FM is capable of. PVM's bandwidth is 37.5MB/s for 32KB messages.

Finally, we comment on SHMEM's bandwidth. As expected, SHMEM Put's bandwidth exceeds FM's. However, for messages between 1–8KB, Push FM's bandwidth is almost identical to SHMEM Put's. Unlike Push FM, however, SHMEM Put does not suffer from cache effects for messages larger than 8KB. Therefore, its bandwidth does not drop after reaching its peak of 120.0MB/s for 32KB messages. And because SHMEM Put does no buffer management, its fixed overhead is reduced enough to give it an 8-byte message bandwidth of 10.5MB/s. SHMEM Get's bandwidth is less than SHMEM Put's, and even less than Push FM's for message sizes larger than 32 bytes. However, it does perform better than Pull FM. While SHMEM Get's 8-byte bandwidth is a reasonably high 6.2MB/s, remote loads limit its bandwidth to 35.3MB/s for 32KB messages.

In summary, Figure 4 demonstrates that while FM supplies PVM-like delivery guarantees, it does so at a cost comparable to the raw data movement cost (SHMEM Put). While it may appear that Pull FM is unnecessary—Push FM exhibits superior latency and bandwidth—it performs much better that Push FM in the presence of heavy network traffic, especially when communication patterns are irregular. (See [29] for details). The Concert runtime [30], for instance, uses exclusively Pull FM because the communication irregularity inherent to Concert programs makes communication robustness dominate overall performance more than baseline latency or bandwidth.

The two implementations of FM on the T3D demonstrate the advantages of a well-defined communication interface, and that the FM interface can be implemented with high performance. The FM interface's design enabled two distinct implementations which, while suitable for different application and network behavior, can be used interchangeably. Because the two implementations have significantly different performance characteristics, application programs benefit from selective use of the two libraries—even within the same program. The performance of the two T3D FM implementations demonstrate that the FM interface need not incur significant overhead. Communication performance delivered by FM is close to the maximum achievable by the underlying hardware.

5 Myrinet FM Implementation

Our workstation cluster implementation of FM utilizes Myricom's 640 Mbps switched network [4] and a collection of Sun SPARCstation workstations [40]. We first discuss the hardware structure of Myrinet. The network exhibits per-hop latencies of around $0.5\,\mu$s. Packets are wormhole-routed [15], so if an output port is busy, the packet is blocked in place. Packets blocked for greater than 50 ms are dropped. In place of the DTB annex on T3D nodes' memory bus, the Myrinet card has a programmable CPU called the LANai,[4] which has 128 KB of SRAM and attaches to the SPARCstation's I/O bus (SBus). The LANai contains three DMA engines—one that transfers between the LANai and host memory, and two that transfer between the network and the LANai memory. The host and the LANai communicate through sharing of LANai memory (mapped into the host's address space) or host memory (via DMA to or from the LANai). While the FM interface for the Myrinet implementation is identical to that on the T3D, the differences in hardware structure dictate a substantially different implementation.

5.1 FM on the Myrinet

All FM implementations must address two key issues. First, the FM layer must efficiently provide reliable, in-order delivery. Second, the implementation must decouple the network and processor. In fact, decoupling is particularly important in workstation clusters, which often operate without coordinated process scheduling. It might be tens or even hundreds of milliseconds before a receiver process is scheduled. These challenges are exacerbated by several additional hardware constraints:

- The LANai processor is ≈ 20 times slower than the host processor,
- The LANai has insufficient memory to effectively buffer messages—only 1.5 milliseconds' worth at network speed, and
- DMA to or from host memory requires that the pages be pinned (i.e. marked unswappable).

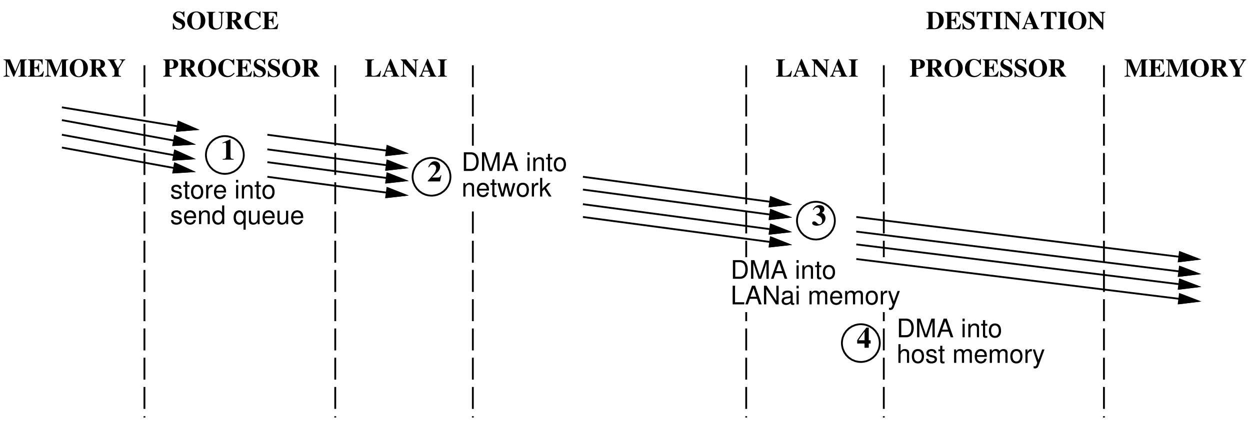


Figure 5: Messaging on the Myrinet

The design of Myrinet FM ensures reliable, in-order delivery with end-to-end window-based flow control and FIFO queueing at all levels. FM uses three queues: a send queue in the LANai,

[4]Our particular Myrinet cards contain version 3.2 LANais, a prototype of the LANai 4.0 processor, but with less memory bandwidth and more limited clock speeds (<25 MHz).

a receive queue in the LANai (used as a staging area), and a larger receive queue in the host's memory (in a pinned-down region). Message transmission consists of four steps, as illustrated in Figure 5. In Step 1, the host processor uses double word stores to move the message data directly into the send queue (in LANai memory). Processor I/O eliminates the cost of copying the data into a pinned DMA-able buffer accessible by the network interface.[5] The outgoing packet is detected by the LANai and DMAed into the network (Step 2). The incoming packet is then detected by the remote LANai and DMAed into local memory (Step 3). Finally, message data buffered in the LANai memory is DMAed to the host, where it can be directly accessed with high efficiency by the receiving process (Step 4). This last step ensures good bandwidth, quick delivery of the packet, and prompt draining of the network, even when the host is busy or not available (i.e. if the FM host program is descheduled).

Myrinet FM's end-to-end window flow control scheme ensures that no packets are lost due to buffer overflow. Each sender is allocated a fraction of a receiving node's host memory queue size (credits), and the sum of the allocated fractions cannot exceed the total storage in the queue. Thus, even if a process neglects to remove data from the network, eventually all of the senders will stop, having used up their credits (i.e. filled their allocated buffers) and no message data will be lost. Furthermore, because FM decouples the host and the network, host unresponsiveness will not prevent messages from being drained from the network. Thus, the Myrinet 50 ms timeout never occurs and therefore, packets are never dropped. Whenever a receiver does remove a message from the queue, however, it sends a credit to the sender[6] to recycle the buffers. Of course, since the network and all of the queues in the sender and receiver are FIFO, message ordering is preserved.

5.2 Myricom's API

Because there are many designs for low-level messaging layers, we describe the Myricom's "Myrinet API" to provide a contrast to FM. The Myrinet API offers a comparable level of interface to FM and, like FM, is also a combination of LANai and host code. The Myrinet API structure is similar to that of traditional networking interfaces for Ethernet. We focus on three significant differences between FM and the Myrinet API. First, in the Myrinet API, the host and the LANai communicate through command FIFOs (e.g. the host tells the LANai it has a packet to send by writing a send command into a queue). Using command queues requires additional synchronization between the host and the LANai, increasing the overhead for communication. Second, the Myrinet API uses DMA on the send side. This requires an extra memory copy (to move the data from the user's data structures into the pinned memory region) and some host-LANai synchronization, which increases overhead. FM uses programmed I/O on the send side in order to eliminate this overhead. Third, the Myrinet API does not guarantee reliable, in-order delivery. It is responsibility of the user to allocate receive buffers in a timely manner (coupling the processor and network) or message data may be lost. This range of differences produces significantly worse latency characteristics for the Myrinet API as shown below.

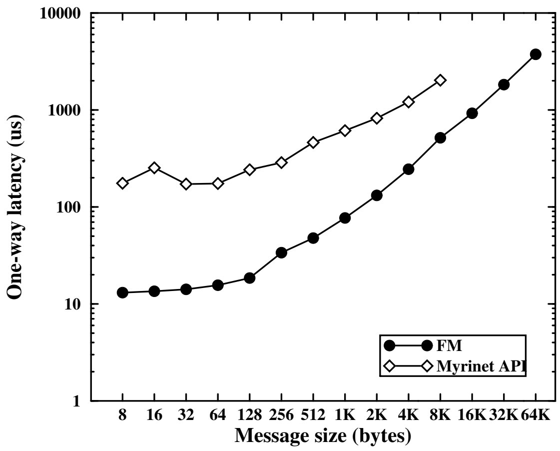
5.3 Performance

To place Myrinet FM's performance in perspective, we compare it with that of the Myrinet API. Although FM provides much stronger guarantees (reliable, in-order delivery), it still delivers su-

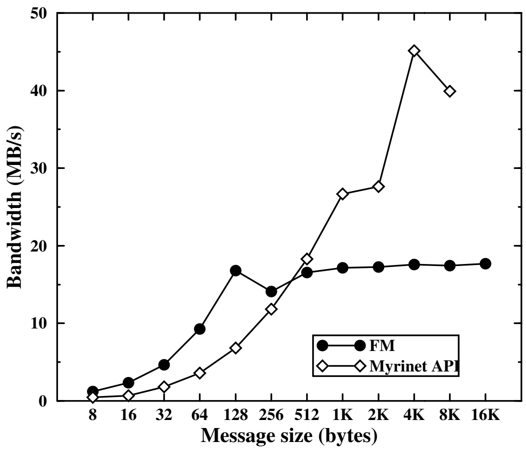
[5] In all of our SBus-based SPARCstation 20, processor I/O cannot use burst transactions mode. Hence, this solution improves latency at the expense of bandwidth. We expect this penalty will be eliminated in future machine designs—particularly those based on PCI—thereby doubling or tripling FM's peak bandwidth.

[6] We aggregate credit messages and—when possible—piggyback credit on ordinary messages for greater efficiency.

perior performance. Figure 6 illustrates the one-way latency and bandwidth of FM running on a Myrinet network containing a pair of SPARCstation 20s with 75 MHz SuperSPARC-II processors and 1 MB level 2 cache.



(a) Latency



(b) Bandwidth

Figure 6: FM 1.1 Performance vs. Myricom's API

The Myrinet FM 1.1 implementation uses 128-byte fixed-size packet frames. For messages > 128 bytes, segmentation and reassembly (SAR) is required, which produces some slight performance anomalies in the transitional region around that message size. Hence, the latency curve is flat up to 128 bytes and then increases linearly with message size. FM's bandwidth peaks at 17.5 MB/s, limited by the I/O bus bridge implementations. This limit disappears with I/O bus bridges that support write aggregation (i.e. exploit bus burst mode for programmed I/O), such as Pentium Pro systems with PCI buses. As in the T3D version, Myrinet FM's performance approaches the maximum possible by the hardware (in the SPARCstation's case, $\approx$ 23 MB/s for programmed I/O bandwidth over the bus bridge). That shows that FM delivers portable, high performance.

The Myrinet FM implementation demonstrates that our high performance low-level messaging layer can be supported portably on rather different hardware. Because the underlying hardware does not provide all of the needed guarantees, our FM implementation includes some LANai firmware and a host processor library which together implement the requisite service guarantees. The cost for supporting these guarantees is not excessive, as demonstrated by the high absolute performance of Myrinet FM and that the resulting performance is significantly better than that of the traditional organization presented by the Myrinet API, especially for small-to-medium sized messages.

6 Applications

For a low-level messaging layer to be successful, it must not only be fast, but also be able to deliver a significant amount of its performance to higher-level messaging layers and, ultimately, to applications. The previous sections focused on the design and performance of FM implementations, addressing in detail the performance of FM. In this section, we first describe the distribution of FM (and related software) and then discuss experiences building communication layers atop FM.

The FM software[7] has been widely disseminated and used in academic, national laboratory, and commercial environments. The Cray T3D version is installed and used at large T3D sites such as PSC, JPL, and LANL. The Myrinet version is installed at far too many sites to list. Federal laboratories users of FM include: NASA (Langley Research Center, Jet Propulsion Laboratory, Center of Excellence in Space Data and Information Sciences at Goddard Space Flight Center), National Laboratories (Los Alamos, Pacific Northwest, Argonne, Lawrence Livermore, Sandia, and the National Energy Research Supercomputing Center). Academic and commercial users are also numerous—over 150 distinct users have registered and downloaded at least one version of our FM software in the past 12 months.

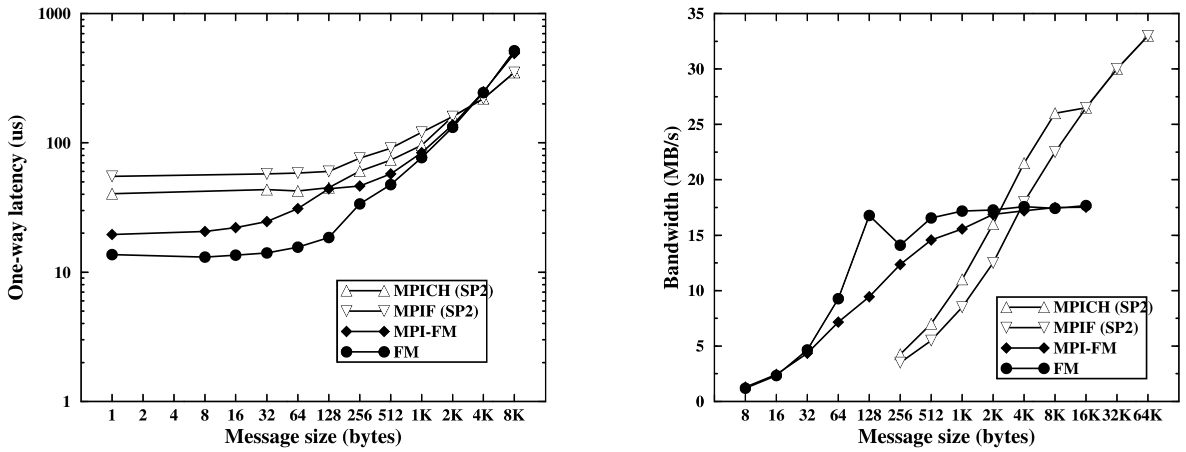
A successful low-level messaging layer must enable the construction of efficient higher level messaging software. To evaluate our FM interface design from this perspective, we have built two higher level messaging interfaces atop FM: the Message Passing Interface (MPI) [35] and UNIX sockets [33]. MPI is a standard of increasing importance for both parallel and distributed message-passing programs. Sockets are a long-standing standard for building multiprocess and, particularly, client-server applications. We first consider the performance of our implementation of MPI atop FM (MPI-FM). Our implementation is a port of the MPICH [23] implementation of MPI. However, we have performed significant restructuring of the code to reduce both buffer copies and control path overhead. Details of the restructuring costs can be found in [32].

The data illustrated in Figure 7 indicates that the high performance of the underlying network hardware is indeed made accessible to higher level messaging layers through FM. While MPI imposes some additional processing overhead—which reduces performance for short messages—we have worked hard to minimize this effect, and near-peak network performance is achieved for messages of modest size—as small as 256 bytes [32]. The additional latency that the MPI processing overhead incurs is only $\approx 5\,\mu$s, which is modest compared to many other implementations [17]. For comparison, performance data for the IBM SP2's two versions of MPI is also included in Figure 7. "MPICH (SP2)" represents a port of MPICH to the SP2, and "MPIF (SP2)" represents an MPI implementation written and optimized specifically for the SP2. Figure 7(a) shows that MPI-FM exhibits superior latency and bandwidth to both of the SP2 MPIs for messages smaller than 4 KB. At that point, MPI-FM's performance levels off due to the limited programmed I/O bandwidth over the SBus bridge.

We are also implementing the UNIX sockets interface, building a user-level library which is linked to an application program and makes calls to the underlying FM implementation. Preliminary FM Sockets performance numbers (Figure 8) indicate a much larger control overhead than the MPI-FM implementation. The major reason for this is the rather complex means by which sockets implementations exchange data with applications and the network. Because sockets implement byte streams without clear message boundaries,[8] both applications and the socket layer itself cannot easily optimize buffer allocation and data movement. The latency numbers reflect the additional control overhead dealing with irregular and unpredictable amounts of data. Also,

[7] FM is available from `http://www-csag.cs.uiuc.edu/projects/comm/fm.html`,

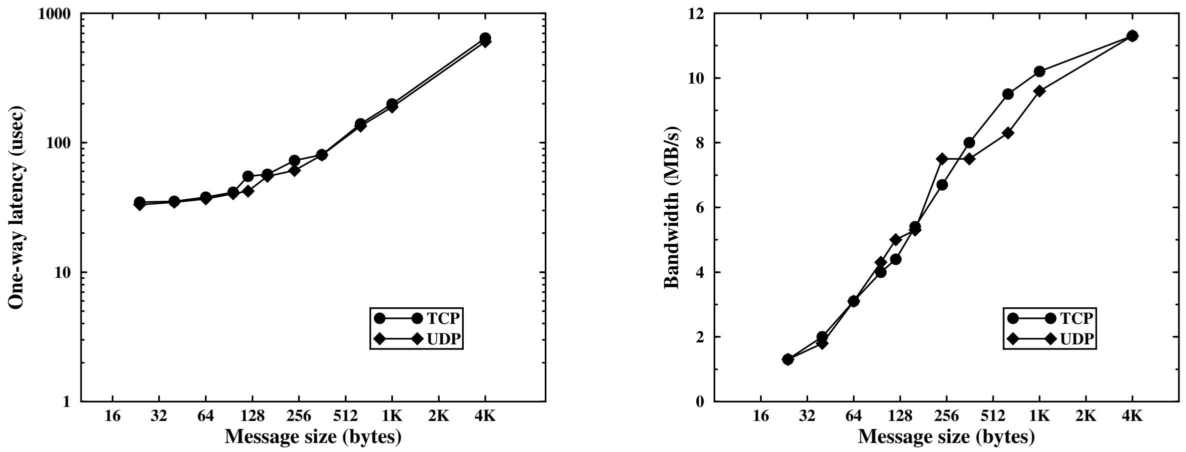
[8] While some sockets implement datagrams, byte streams are the general case.



(a) Latency

(b) Bandwidth

Figure 7: MPI-FM Performance



(a) Latency

(b) Bandwidth

Figure 8: FM Sockets Performance

FM Sockets does not achieve raw FM's bandwidth of 17.5 MB/s, because FM's receive queue needs to be marked uncacheable to ensure consistency when using hyperSPARC CPUs (as were used in Figure 8). Hence, the cost of segmentation and reassmebly is greatly increased. Given those limitations, FM Sockets achieves good performance, partly due to the fact that FM already provides reliable, in-order delivery, thereby obviating the need to implement it in the socket layer. This shows that because FM provides important guarantees, higher-level messaging layers can achieve performance approaching FM's.

To put the FM Sockets performance in perspective, we compare it to the performance of Myricom's TCP and UDP implementations. According to third-party measurements [31], Myricom's TCP has an 8-byte one-way latency of 751.0 μs and their UDP has an 8-byte one-way latency of 721.5 μs. FM Sockets' performance is an order of magnitude better—only 34.8 μs for TCP and 33.3 μs for UDP. Furthermore, Myricom's TCP peaks at 9.4 MB/s, versus 11.3 MB/s for FM Sockets, and their UDP peaks at 9.8 MB/s, again versus 11.3 MB/s for FM Sockets. Note that the above is not a fair comparison. Myricom implemented TCP and UDP as standard, kernel-level drivers, while FM Sockets runs entirely user-level. Still, it illustrates the performance benefits of implementing sockets on top of FM.

From our experience building messaging layers, we have been able to design FM to make high performance accessible to higher-level messaging layers. However, while we have achieved good performance, the design of the MPI and UNIX sockets interfaces has given insight into several additional ways to improve the FM interface. These ongoing enhancements are discussed in Section 7.

7 Discussion

We have demonstrated that our FM implementations achieve excellent performance while maintaining a portable interface across widely differing platforms—MPPs and NOWs. However, the FM interface is still evolving. From our experience implementing MPI-FM and FM Sockets, we gained insight into FM's benefits and shortcomings. Specifically, we learned that the lack of gather-scatter functionality and the lack of flow control between FM and higher-level messaging layers can significantly affect the overall performance of the system.

Gather-scatter is needed primarily for attaching and removing headers to messages. Generally, each messaging layer in a hierarchy adds a header to the data it is passed and then passes the header+data to the next messaging layer. On the receiving side, each messaging layer strips off its corresponding header before passing the remaining data upwards. Because FM can send only contiguous data, messaging layers built on top of FM must write a header into a region of memory and copy the data they need to send into the space immediately following that prior to calling `FM_send()`. Measurements revealed that MPI-FM was spending a significant amount of time doing this data copy. We discovered that by eliminating this copy with even a simple two-piece gather, performance improved significantly [32]. Similarly, we found that on the receive side, stripping the message header can be made faster by adding scatter functionality.

Flow control between FM and FM Sockets was another problem we observed. In the current FM interface, `FM_extract()` extracts all the available messages and calls all the corresponding handlers in sequence. The problem with FM Sockets is that only one receive is posted between successive calls to `FM_extract()`. Hence, when `FM_extract()` is called, at most one pending message has a buffer preallocated. The rest must be buffered until more receives are posted. Allocating buffer space, copying the data from FM's buffers to FM Sockets' buffers, and eventually copying the data into user-provided buffers are all expensive operations. A solution is for FM to provide a way to pace the network, that is, to limit the amount of data `FM_extract()` extracts.

Sidebar C describes FM 2.0, which uses a modified FM interface that supports high-level flow control and an elegant form of gather/scatter.

8 Related Work

FM provides a portable interface for high performance communication, and numerous efforts have been made in this area. We briefly survey efforts in hardware architectures and more recently software interfaces which bring high performance communication to applications.

Hardware architects have long pursued high performance for short messages by integrating the network interface with the CPU itself [5, 16, 25, 45]. Innovations included interfacing the network to the instruction set and register file, message-field based dispatch, communication driven scheduling, etc. However, none of these innovations persisted into mainstream high performance processors because of their requirement of changes deep in the processor. Also, these innovations did not address issues of network and output contention such as Pull FM is designed to tolerate.

Less aggressive parallel architectures use commodity microprocessors and add external communication logic (e.g. the TMC CM-5 [42] and Cray T3D [14]). These machines are quite similar to workstation clusters based on, for example, Myricom's Myrinet [4] or DEC's Memory Channel [21]. Work on Active Messages [44] bridged the gap between the aggressive integration of communication into the processor and these hybrid systems, showing how much of the communication performance benefit could be reaped on commodity microprocessor based systems. Active Messages crystallized a basic low-level communication model which is embodied in many communication layers, including Fast Messages. As discussed further in Sidebar B, FM adds ordering, buffering, and flow control guarantees to the foundation established by that work. FM's efficient portability across a range of parallel systems—the Cray T3D (passive interface and remote memory access) and Myrinet-based clusters (communication coprocessor and private memory)—demonstrate FM's portability.

A number of recent efforts have explored novel low-level communication interfaces. U-Net [43] provides buffer management, but no flow control, implying that data can be lost due to rate mismatch. The low-level interface supporting the Hamlyn [8] architecture provides flow control, but no buffer management, implying that higher-level messaging layers must provide their own buffer-management routines to prevent data from being overwritten. FM provides a more complete solution to ensuring message delivery. By removing the responsibility for flow control and buffer management from higher-level messaging layers, the FM interface permits platform-specific implementations—hence, optimizations—of flow control and buffer management. The Real World Computing Partnership's PM [41], like FM, runs on Myrinet-connected workstations and performs both flow control and buffer management. However, the PM implementation uses an optimistic flow control mechanism and variable-sized packets. We are currently exploring opportunities to make a detailed comparison.

9 Summary

Illinois Fast Messages (FM) is a high-performance, low-level messaging layer. It differentiates itself by providing a portable interface and by "bridging the performance gap"—enabling messaging layers built on top of FM to deliver much of FM's performance.

Often, there is a tradeoff between portability and performance. For FM, however, we identified the key similarities between MPPs and NOWs (present and future) and included in the interface only those features that are generally useful and that can be implemented efficiently on both types of systems. That is, we made the FM interface as portable as possible, up to the point where performance would have been sacrificed. FM implementations not only deliver high performance, but also provide three key guarantees that enable streamlined implementations of higher-level protocols atop FM. These guarantees are:

- Reliable delivery,
- In-order delivery, and
- Decoupling of processor and network.

To truly exploit high-performance computing on MPPs and NOWs at the application level, low-level message layers should provide all of those, and FM does. FM supplies reliable, in-order delivery by providing buffer management and flow control and uses buffering to decouple the processor and the network. To demonstrate that these features are portable and remain efficient across implementations, we have implemented the FM interface on both the Cray T3D and a Myrinet network of SPARCstations. Our multiple implementations of FM show that it is possible for a messaging layer to achieve a sizable fraction of the hardware performance without short-changing the layers above it. All of our implementations exhibit excellent latency—as low as 6.1 μs on the T3D and 13.1 μs on Myrinet—and bandwidth—as much as 112.9 MB/s on the T3D and 17.5 MB/s on Myrinet. FM on the T3D performs better than the vendor's PVM implementation and comparably to the lower-level, more primitive SHMEM. FM on Myrinet performs better than the vendor's API and, furthermore, its performance is quite competitive with that of current MPPs, such as the SP2.

While there are many high-speed messaging layers, many of them achieve good performance by relegating costly operations to higher-level layers. Although doing so makes the low-level layer *appear* fast, it generally has a detrimental affect on actual, application-level performance, because higher-level layers must add the missing functionality, often at a higher price than the lower-level layer would have paid. FM sacrifices a tiny amount of its raw performance [37] for a large payoff to the overall performance of the system. By providing generally useful service guarantees such as reliable, in-order delivery, FM saves messaging layers and applications built on top of it from the burden and performance penalty of adding those features. Hence, not only is FM capable of low-latency and high-bandwidth communication, but so are the higher-level messaging layers built on top of it. We validated this claim with an implementation of UNIX sockets and MPI on top of Myrinet FM—currently the fastest MPI available for a cluster and competitive even against MPI implementations running on current MPPs.

FM software distributions and the latest information about the FM project are available from `http://www-csag.cs.uiuc.edu/projects/comm/fm.html`.

10 Future Work

While FM demonstrates that portable high performance is achievable, there are still a variety of challenging research questions to be explored. Since doing the work described in this paper, we have found new techniques for delivering more performance to higher-level messaging layers. Sidebar C describes the FM 2.0 interface, which contains mechanisms for pipelining data transfer across the network, reducing memory copies needed for higher-level protocol processing, and increasing communication efficiency in the context of more extreme decoupling.

While the original goal of FM evolved was to support more efficient parallel computation, such high-speed communication systems can have a much broader impact on distributed computing. The additional requirements of such a system include:

- support for multiple users and processes per node,
- coordinated scheduling,
- integrating separately-managed computing domains, and
- interoperability with traditional networks and networking protocol structures.

Distributed systems typically run multitasking operating systems on each node. This implies that if multiple, independent processes on the same node want to use the network concurrently, the communication subsystem must prevent inadvertent interactions. Messaging layers such as FM should therefore provide protection between processes as well as a way to demultiplex messages to the appropriate process on a destination node. The challenge in doing so efficiently involves a complex tradeoff between dedicating resources and inline interpretation overhead.

Because each node runs its operating system independently from other nodes, communicating processes are often not scheduled simultaneously. This lack of coscheduling can drastically increase response time—often into the tens-of-milliseconds range—and thereby severely reduce performance. Gang scheduling is inappropriate in the context of multiuser, multitasking operating systems because it requires scheduling decisions to be globally agreed upon. This is costly and scales poorly. A better approach is to integrate communication and scheduling. That is, FM and the operating system should work together to coschedule communicating processes by exploiting local process information and information about message arrival and launch. We have already implemented a version of FM that does exactly that [38], but our implementation is limited to a single communicating thread per node. Ongoing work involves generalizing the algorithm and implementation to support an arbitrary number of threads and processes per node.

Among the challenges in converting from a strictly parallel to an integrated parallel and distributed model is providing support for large-scale computing—computing on systems with more nodes than fit in a single administrative domain. For such systems, communication layers support extremely large numbers of nodes. Again, this entails challenges in resource management. For example, FM's static window-based flow control scheme might need to be replaced with something more dynamic.

Finally, there is much networking code in existence, and low-speed networks are still numerically dominant. To incrementally integrate high speed communication systems into existing networks running legacy software, these systems must interoperate. It must be possible for messaging layers such as FM to interact with established networking protocol structures without sacrificing an inordinate amount of performance on high-speed networks.

Acknowledgements

The authors would like to acknowledge Mario Lauria for producing MPI-FM and Olga Natkovich for producing FM Sockets. In addition, we thank Xingbin Zhang and Matt Buchanan for their comments on improving the presentation of this material.

The research described in this paper was supported in part by DARPA Order #E313 through the US Air Force Rome Laboratory Contract F30602-96-1-0286, NSF grant MIP-92-23732, ONR grants N00014-92-J-1961 and N00014-93-1-1086, NASA grant NAG 1-613, and supercomputing resources at the Jet Propulsion Laboratory. Support from Intel Corporation, Tandem Computers, Hewlett-Packard, and Motorola is also gratefully acknowledged. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809. Vijay Karamcheti is supported in part by an IBM Computer Sciences Fellowship.

A Cray T3D and Myrinet Hardware

One node of the Cray T3D is diagrammed in Figure 9. The three main components of a node are the processing elements,[9] the block transfer engine (BLT), and the network interface. The processing element contains an Alpha microprocessor, local memory, and some communication-specific logic. The parts of the node that are the most relevant to FM are the data translation buffer (DTB) annex, which is needed to set up a logical connection with other nodes, the fetch-and-increment unit, which is used for push messaging, and the atomic swap unit and prefetch queue, which are used for pull messaging,

Figure 10 depicts Myricom's Myrinet interface. Most of the functionality is provided by the LANai, a custom VLSI chip. Only memory (128 KB) and I/O bus logic require additional chips. The LANai CPU is a custom 32-bit processor with special registers for sending and receiving data, starting DMA operations, and checking the status of various operations.

[9] Each node actually contains two processing elements. For clarity, the figure shows only one.

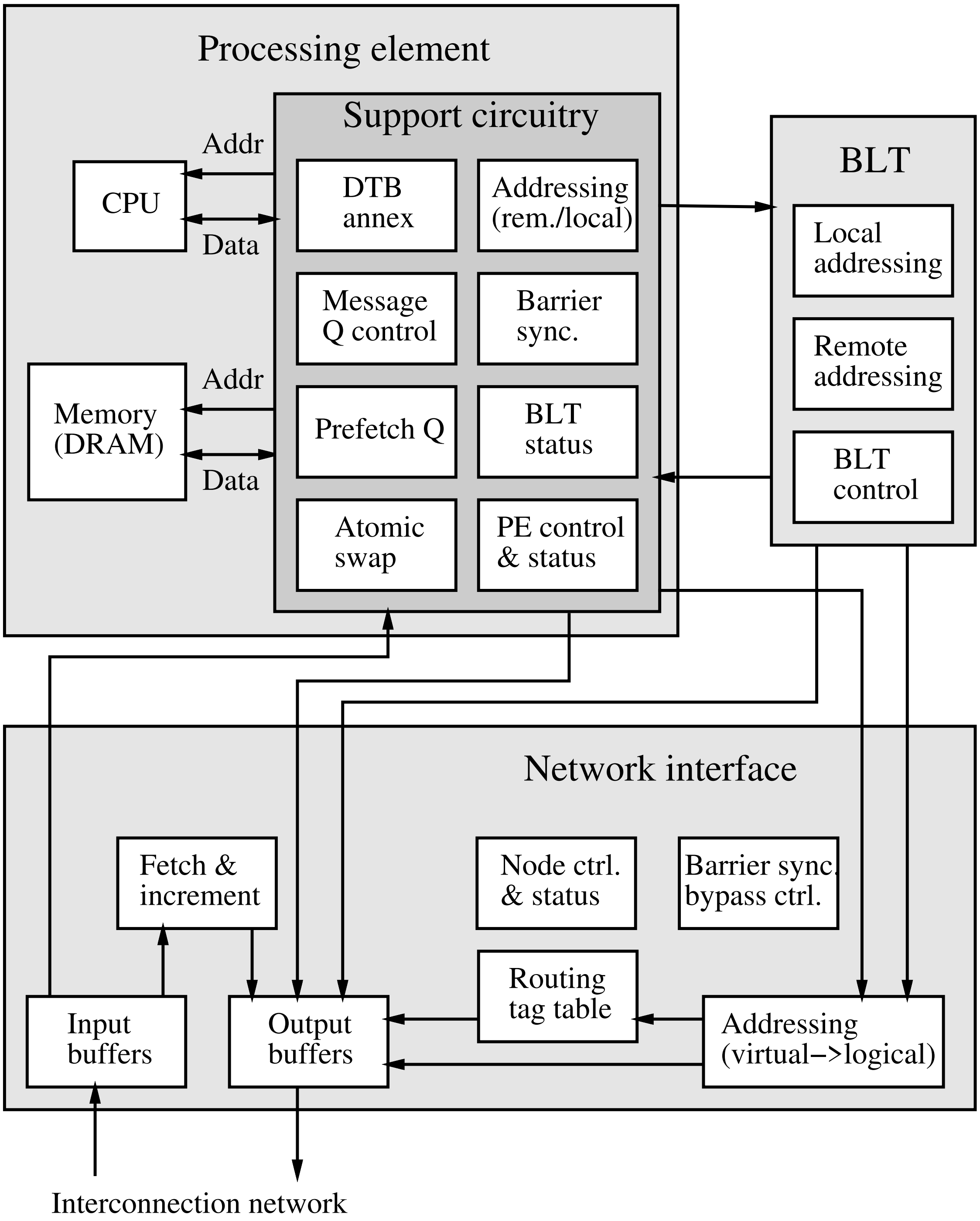


Figure 9: T3D node

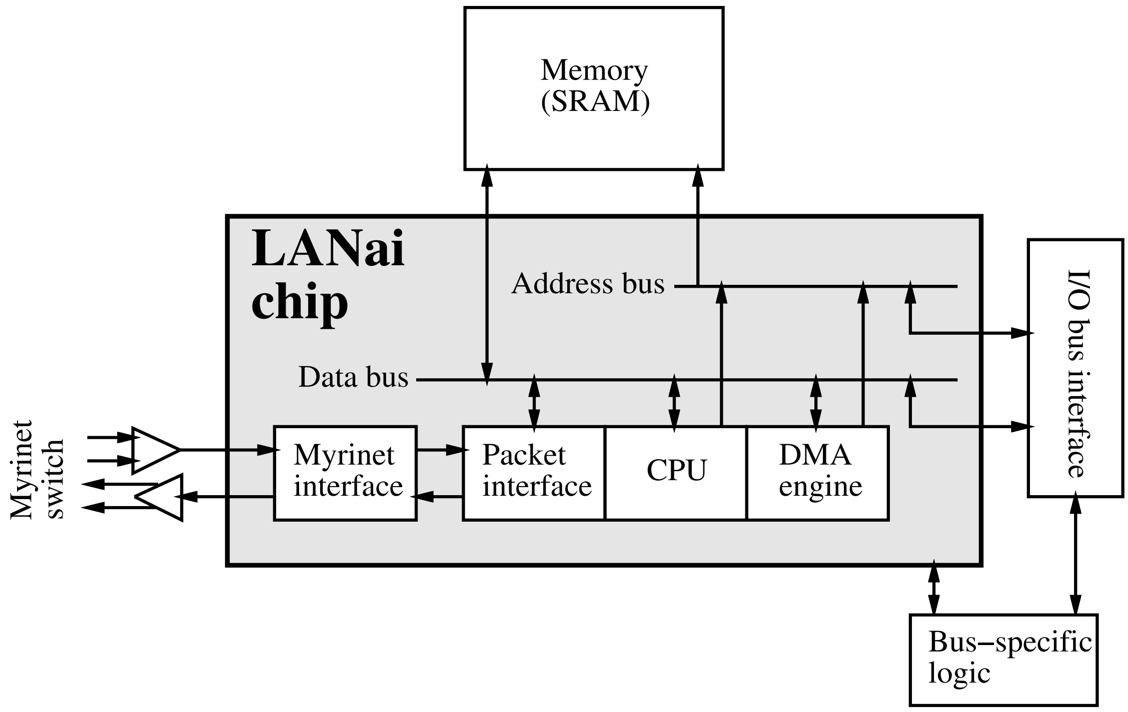


Figure 10: Myrinet interface

B Comparing Fast Messages and Active Messages

Fast Messages builds on the work of others in the area. The original FM 1.1 interface is closely modeled on that of Active Messages (AM) [44], which in turn has as its intellectual antecedents hardware architectures that closely integrated communication and computation in message-driven [16], dataflow [26], and even systolic [5] architectures. We briefly describe the key similarities and differences between the FM 1.1 system and Active Messages. Note that the FM 2.0 systems significantly extend the functionality of the interface, as shown in Sidebar C.

Fast Messages adopts a number of key design choices validated in the design and success of Active Messages. First, small "handlers" are associated with messages, which incorporate incoming data into the ongoing computation. Second, these handlers are user-specified, and can be used to implement protocols atop the primitive communication layer. These similarities give AM and FM a similar API, and, as a result, simple program examples are often nearly identical.

The crucial differences between FM and AM involve data reception, buffering, deadlock avoidance, and message ordering. AM systems include both explicit polling operations and implicit polling operations in each message send. This ensures data is removed from the network, eliminating the possibility of internal network as well as store-and-forward deadlock. In contrast, FM provides an explicit polling operation, but includes no implicit poll in send operations. This subtle difference has two important implications. First, control over when a message is processed through extract placement in FM programs allows control over data locality in the receiver's memory hierarchy. Second, a receiver may not process network messages for a long time, so a modest amount of buffering and flow control is required in FM to eliminate the possibility of deadlock. In contrast, AM systems can benefit from buffering, but do not require it. However, the implicit poll involved in sends does reduce program control over the scheduling of message reception. Finally, FM guarantees that messages are delivered in order, while AM makes no such guarantee.

These differences have several other facets, whose ramifications are only now being played out in research and prototype systems. For instance, the AM approach normally avoids buffering and may therefore more efficiently integrate "direct transfer" operations such as PUT/GET into an implementation or interface. On the other hand, FM's decoupling of the processor and network through a buffered, flow controlled stream of data may present advantages in supporting efficient multitasking or scaling to networks with long latencies. We look forward to the exploration of these issues in future high-performance communication systems.

C Fast Messages Version 2.0

Since the initial submission of this article (June, 1996) we have introduced the FM 2.0 interface. While the FM 1.1 interface supports ordered, reliable delivery and decoupling of communication and computation—a carefully-chosen set of guarantees—in some cases, it can still require higher-level messaging layers to perform memory copies, as it lacks network pacing and gather/scatter functionality.

FM Sockets exposed the need for network pacing. `FM_extract()` is called only when an FM Sockets program posts a receive (i.e. calls `recv()`, `recvfrom()`, or `recvmsg()`). But because `FM_extract()` processes the entire receive queue, FM Sockets is forced to buffer (i.e. copy) a potentially large amount of data. When subsequent receives are posted, FM Sockets must again copy data, this time from the unposted receive buffer into a program-specified location. The FM 2.0 interface reduces such additional memory copying by allowing higher-level messaging layers to pace message processing by specifying the maximum number of bytes that `FM_extract()` processes (Table 2). Thus, rather than processing all available data on a socket receive operation, FM Sockets need process data only until the overlying program's receive request is satisfied. This control also allows FM 2.0 programs to more effectively schedule communication processing with computation (e.g. to amortize communication costs behind computation time).

Function	Operation
`FM_begin_message(`*dest,size,handler*`)`	Open a streamed message
`FM_send_piece(`*stream,buf,size*`)`	Add data to a streamed message
`FM_end_message(`*stream*`)`	Close a streamed message
`FM_extract(`*maxbytes*`)`	Process up to *maxbytes* of received messages
`FM_receive(`*buf,stream,size*`)`	Receive data from a streamed message into a buffer

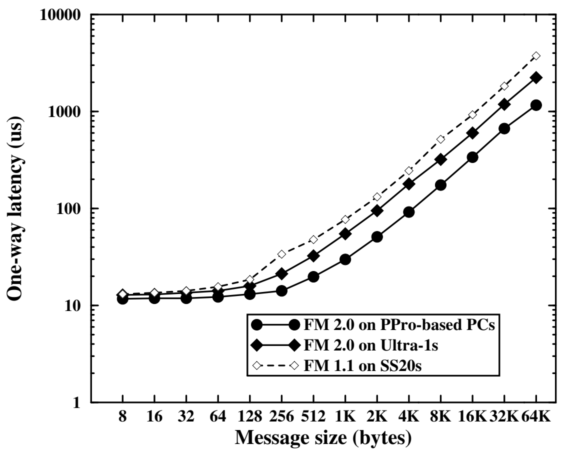
Table 2: FM 2.0 API

MPI-FM validated the importance of gather/scatter—initially deferred to higher-level messaging layers for implementation if needed. MPI-FM adds protocol-specific headers to the beginning of each message. Without gather/scatter built into FM, higher-level messaging layers must copy program data twice: once on the send side to append protocol-specific headers and once on the receive side to strip those headers. These extra copies hurt performance significantly [32]. Traditional gather vectors (lists of <length,offset> pairs sent in sequence) remove copies on the send side, but traditional scatter vectors are insufficient for removing copies on the receive side, as the target location of a message is known only when the message header is parsed.

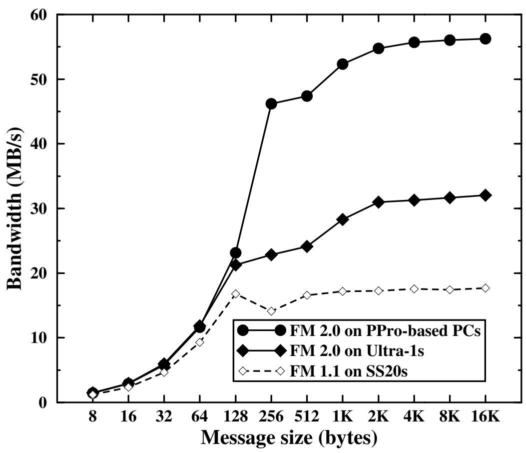
Instead of gather/scatter vectors, FM 2.0 implements gather and scatter using a novel concept called "streaming messages." Message data is written and read piecewise via a stream abstraction. On the send side, a higher-level messaging layer opens a message with `FM_begin_message()`, appends zero or more pieces of data to the message with `FM_send_piece()` (e.g. a header followed by program data), and finally closes the message with `FM_end_message()` (Table 2). On the receive side, a message handler is passed an opaque "stream" object instead of a pointer to a piece of memory. The handler receives data piecewise from the stream object, specifying a target memory location for each `FM_receive()`. Because each message is a stream of bytes (unlike TCP, which does not possess the concept of a message), the size of each piece received need not equal the size of each piece sent, as long as the total message sizes match. Thus, higher-level receivers can examine a message header and, based on its contents, scatter the message data to to appropriate locations. Note that the streaming interface is only made possible because FM provides end-to-end flow control.

In addition to gather/scatter, streaming message interface also provides the ability to pipeline individual messages; message processing can begin at the receiver even before the sender has finished. This increases the throughput of higher-level messaging layers built atop FM.

Although FM 2.0 is strictly more powerful than FM 1.1, there need not be a severe performance penalty. Figure 11 shows the performance of Myrinet FM 2.0 on a pair of Dell Optiplex GXPros (200 MHz Pentium Pro-based PCs) and a pair of Sun Ultra-1 computers.



(a) Latency



(b) Bandwidth

Figure 11: FM 2.0 Performance

References

[1] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve G. Steinberg, and Katherine Yelick. Empirical evaluation of the CRAY-T3D: A compiler perspective. In *Proceeedings of the International Symposium on Computer Architecture*, pages 320–331, June 1995. Available from `http://www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps`.

[2] Ray Barriuso and Allan Knies. *SHMEM User's Guide*. Cray Research, Inc., May 1994.

[3] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceeding of the International Symposium on Computer Architecture*, pages 142–153, April 1994. Available from `http://www.cs.princeton.edu/shrimp/papers/isca94.paper.ps`.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet—a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995. Available from `http://www.myri.com/research/publications/Hot.ps`.

[5] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 70–81. IEEE Computer Society, 1990.

[6] J. Boudec. The Asynchronous Transfer Mode: A tutorial. *Computer Networks and ISDN Systems*, 24:279–309, 1992.

[7] Lawrence S. Brakmo and Larry L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communication*, 13(8):1465–1480, October 1995. Available from `ftp://ftp.cs.arizona.edu/xkernel/Papers/jsac.ps`.

[8] Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes. Hamlyn: A high-performance network interface with sender-based memory management. In *Proceedings of the IEEE Hot Interconnects Symposium*, August 1995. Available from `http://www.hpl.hp.com/personal/John_Wilkes/papers/HamlynHotIntIII.pdf`.

[9] Ramón Cáceres. *Multiplexing Traffic at the Entrance to Wide-Area Networks*. PhD thesis, University of California, Berkeley, December 1992.

[10] Chran-Ham Chang, Richard Flower, John Forecast, Heather Gray, William R. Hawe, K. K. Ramakrishnan, Ashok P. Nadkarni, Uttam N. Shikarpur, and Kathleen M. Wilde. High-performance TCP/IP and UDP/IP networking in DEC OSF/1 for Alpha AXP. *Digital Technical Journal*, 5(1), Winter 1993. Available from `ftp://ftp.digital.com/pub/Digital/info/DTJ/nw-06-tcp.ps`.

[11] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proceedings of SIGMETRICS/PERFORMANCE*, May 1995. Available from `http://www-flash.stanford.edu/OS/papers/SIGMETRICS95/numa-os.ps.Z`.

[12] Andrew Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang. Supporting high level programming with high performance: The illinois concert system. In *Proceedings of the Second International Workshop on High-level Parallel Programming Models and Supportive Environments*, April 1997.

[13] Bilal Chinoy and Kevin Fall. TCP/IP and HIPPI performance in the CASA gigabit testbed. In *Proceedings of the 1994 USENIX Symposium on High-Speed Networking*, August 1994.

[14] Cray Research, Inc. *Cray T3D System Architecture Overview*, March 1993.

[15] W. J. Dally and C. Seitz. The torus routing chip. *Distributed Computing*, pages 187–196, 1986.

[16] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills. Architecture of a message-driven processor. In *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, pages 189–196. IEEE, June 1987.

[17] Jack J. Dongarra and Tom Dunigan. Message-passing performance of various computers. Technical Report CS-95-299, University of Tennessee, August 1995. Available from `http://www.netlib.org/tennessee/ut-cs-95-299.ps`.

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

[19] G. Fox et al. *Solving Problems on Concurrent Processors*, volume I and II. Prentice-Hall, 1988.

[20] G. Geist and V. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Proceedings of the Sixth Distributed Memory Computers Conference*, pages 258–61, 1991.

[21] Richard B. Gillett. Memory Channel network for PCI. *IEEE Micro*, 16(1):12–18, February 1996. Available from `http://www.computer.org/pubs/micro/web/m1gil.pdf`.

[22] Lee Goldberg. 100Base-T4 transceiver simplifies adapter, repeater, and switch designs. *Electronic Design*, pages 155–160, March 1995.

[23] William Gropp and Ewing Lusk. User's guide for `mpich`, a portable implementation of MPI. Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratory, Mathematics and Computer Science Division, 1996. Available from `http://www.mcs.anl.gov/mpi/mpiuserguide/paper.html` and `ftp://info.mcs.anl.gov/pub/mpi/userguide.ps.Z`.

[24] Riccardo Gusella. A measurement study of diskless workstation traffic on Ethernet. *IEEE Transactions on Communications*, 38(9), September 1990.

[25] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 111–122, Boston, Massachusetts, October 1992. Available from `ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-342.ps.gz`.

[26] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance studies of Id on the Monsoon dataflow system. *Journal of Parallel and Distributed Computing*, 18(3):273–300, July 1993. Available from `ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-345-3.ps.gz` or `http://www.csg.lcs.mit.edu:8001/monsoon/monsoon-performance/monsoon-performance.html`.

[27] A. Jinzaki, T. Ni'inomi, and S. Kobayashi. Illinois Fast Messages on 1 Gbps Fibre Channel. In the *ASPLOS-VII NOW/Cluster Workshop*, Cambridge, Massachusetts, October 1996. Available from `http://www-csag.cs.uiuc.edu/individual/achien/asplos/posters/fm-fibrechannel.ps`.

[28] Vijay Karamcheti and Andrew Chien. Software overhead in messaging layers: Where does the time go? In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994. Available from `http://www-csag.cs.uiuc.edu/papers/asplos94.ps`.

[29] Vijay Karamcheti and Andrew A. Chien. A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D. In *Proceedings of the International Symposium on Computer Architecture*, 1995. Available from `http://www-csag.cs.uiuc.edu/papers/cm5-t3d-messaging.ps`.

[30] Vijay Karamcheti, John Plevyak, and Andrew A. Chien. Runtime mechanisms for efficient dynamic multithreading. *Journal of Parallel and Distributed Computing*, 37:21–40, 1996.

[31] Kimberly K. Keeton, Thomas E. Anderson, and David A. Patterson. LogP quantified: The case for low-overhead local area networks. In *Hot Interconnects III: A Symposium on High Performance Interconnects*, 1995. Available from `http://now.cs.berkeley.edu/Papers/Papers/hotinter95-tcp.ps`.

[32] Mario Lauria and Andrew Chien. MPI-FM: High performance MPI on workstation clusters. *Journal of Parallel and Distributed Computing*, 1997. To appear; currently available from `http://www-csag.cs.uiuc.edu/papers/mpi-fm.ps`.

[33] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quaterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1988.

[34] Neil R. McKenzie, Kevin Bolding, Carl Ebeling, and Lawrence Snyder. Cranium: An interface for message passing on adaptive packet routing networks. In *Proceedings of the 1994 Parallel Computer Routing and Communication Workshop*, May 1994. Available from `ftp://shrimp.cs.washington.edu/pub/chaos/docs/cranium-pcrcw.ps.Z`.

[35] Message Passing Interface Forum. The MPI message passing interface standard. Technical report, University of Tennessee, Knoxville, April 1994. Available from `http://www.mcs.anl.gov/mpi/mpi-report.ps`.

[36] R. Metcalfe and D. Boggs. Ethernet: Distributed packet-switching for local computer networks. *Communications of the Association for Computing Machinery*, 19(7):395–404, 1976.

[37] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing*, December 1995. Available from `http://www-csag.cs.uiuc.edu/papers/myrinet-fm-sc95.ps`.

[38] Patrick Sobalvarro, Scott Pakin, Andrew Chien, and William Weihl. FM-DCS: An implementation of dynamic coscheduling on a network of workstations. In the *ASPLOS-VII NOW/Cluster Workshop*, Cambridge, Massachusetts, October 1996. Available from `http://www-csag.cs.uiuc.edu/individual/achien/asplos/posters/fm-dcs.ps`.

[39] Craig B. Stunkel and W. Kent Fuchs. An analysis of cache performance for a hypercube multicomputer. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):421–432, July 1992.

[40] Sun Microsystems Computer Company, Mountain View, CA. *SPARCstation 20 System Architecture*, 1995.

[41] Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. Design and Implementation of PM: A Communication Library for Workstation Cluster. In *JSPP*, 1996. (In Japanese).

[42] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264. *The Connection Machine CM-5 Technical Summary*, October 1991.

[43] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner 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. Available from `http://www.cs.cornell.edu/Info/Projects/ATM/sosp.ps`.

[44] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture*, 1992. Available from `http://www.cs.cornell.edu/Info/Projects/CAM/isca92.ps`.

[45] Colin Whitby-Strevens. The transputer. In *Proceedings of 12th International Symposium on Computer Architecture*, 1985.

[46] X3T11 Technical Committee, Project 1119D. *Fibre Channel Physical and Signalling Interface–3 (FC-PH-3), Revision 8.3.* American National Standards Institute, January 1996. Working draft proposed American National Standard for Information Systems. Available from `http://www.network.com/~ftp/FC/PH/FCPH3_83.ps`.