FLIPC: A Low Latency Messaging System for Distributed Real Time Environments

David L. Black, Randall D. Smith,
Steven J. Sears, and Randall W. Dean
Open Software Foundation Research Insititute

FLIPC: A Low Latency Messaging System for Distributed Real Time Environments

David L. Black,[1] Randall D. Smith, Steven J. Sears,[2] and Randall W. Dean

Open Software Foundation Research Institute, Cambridge, MA

dlb@osf.org, randys@osf.org, sjs@novell.com, rwd@osf.org

Abstract

FLIPC is a new messaging system intended to support distributed real time applications on high performance communication hardware. Application messaging systems designed for high performance computing environments are not well suited to other environments because they lack support for the complex application structures involving multiple processes, threads, and classes of message traffic found in environments such as distributed real time. These messaging systems also have not been optimized for medium size messages found in important classes of real time applications. FLIPC includes additional features to support applications outside the high performance computing domain. For medium size messages, our system significantly outperforms other messaging systems on the Intel Paragon. An explicit design focus on programmable communication hardware and the resulting use of wait-free synchronization was a key factor in achieving this level of performance. The implementation of FLIPC was accelerated by our use of PC clusters connected by ethernet or by a SCSI bus as development platforms to reduce the need for Paragon time.

1 Introduction

Messaging systems for high performance computing are designed to deliver as much of the hardware performance of the interconnect to applications as possible. Examples include the NX system on the Intel Paragon [11], CMMD on the Thinking Machines CM-5 [17], and Active Messages on various platforms [2][19]. These systems have been optimized for numerical supercomputing to the detriment of their use in other environments. Among the factors that contribute to this limitation are the assumption of a single threaded application structure, system optimizations for small and large message sizes to the detriment of intermediate sizes, and the assumption of a single application environment in which there are no competing sources of message traffic. Our work concerns distributed real time environments in which these assumptions do not hold.

Continuing advances in networking technology, such as ATM [18], and Myrinet [1], have dramatically increased the communication performance available to environments other than numerical supercomputing. This paper describes our experimental implementation of a high performance messaging system for one such environment, namely event driven distributed real time. This environment is characterized by the need to support multiple application threads of varying importance or priority, and messaging streams of varying importance concurrently on each node. This includes not only processing, but also resource allocation. For example, the system must not only process a message announcing detection of an incoming message in preference to a message indicating that it is time for preventative maintenance, but must also ensure that the latter message does not consume resources required to handle the former.

The event driven nature of applications in this environment results in messages that fall between the small and large sizes found in numerical supercomputing. The events cannot be described by very small messages, and aggregation of events into larger messages is limited by the impact of the aggregation delay on system response. Examples of such systems include distributed systems for process control, factory floor automation, and military command and control [5] (e.g., AEGIS, AWACS).

1. This research was supported in part by the Advanced Research Projects Agency (ARPA) and the Air Force Material Command (AFMC).

2. Steven J. Sears is currently with Novell, Inc.

The FLIPC messaging system is designed for event driven real time environments. Among its more important characteristics are:

- performance optimization for medium sized messages (50-500 bytes),
- support for multithreaded applications,
- support for threads and message streams of varying importance,
- support for explicit control of resource allocation.

2 Communication Interface Architecture

FLIPC is designed to leverage the powerful programmable controllers that are increasingly found in the interfaces to high speed interconnects and networks. Examples of these controllers include the dedicated message processor on each node of a Paragon system [6], embedded microprocessors in communication controllers such as the i960 [4], and custom designed communication controllers such as Myrinet [1], SCSI [10], and FLASH [8]. The increasing use of such controllers is motivated by the need to match increases in processor speed with increases in communication performance, and the resulting desire to off-load communication functionality from the main processor(s). Similar trends have been observed in other domains, such as mainframe input/output and graphics controllers[14]. Experience in these domains suggests that the power and functionality of the programmable controllers in network interfaces will continue to increase.

The power of these interface controllers cannot be directly utilized by applications because the controllers and their execution environment are functionally specialized to communication tasks. The resulting obstacles to execution of application code include:

- **Instruction Set Differences:** Of the controllers cited above, only the dedicated Paragon processor employs the same instruction set as the application processor(s).
- **Protection Concerns:** Controllers have access to critical system resources, often including all of physical memory. Communication is also critical to system operation, as a controller crash may imply a node crash. A related problem is that an application must be prevented from monopolizing the communication resource to the exclusion of other applications. As a result, untrusted code cannot be executed on such interface controller.
- **Memory Access Restrictions:** Neither the SCSI nor Myrinet controllers are capable of performing read-modify-write atomic operations such as test-and-set on main memory.
- **Execution Restrictions:** A common structure for controller software is a non-preemptible event loop. This puts restrictions on the time that may be consumed by added code; excessive consumption may have undesirable side effects on unrelated communications.

FLIPC's architecture takes advantage of interface controllers by programming them as operating system components rather than as part of the applications. This enables the use of system programming tools that target the embedded controller, and simplifies protection concerns. Restrictions on memory access and execution are dealt with in the design of the FLIPC component that executes on the controller. The alternate approach of using software fault isolation [20] is of limited utility because correctness of the code executing on the controller depends not only on the memory it accesses, but also on completing execution in a timely fashion. Restrictions such as forbidding backwards branches are necessary, greatly reducing the programming flexibility.

3 Message Sizes

The design of FLIPC is based on the observation that there are three distinct message size classes with corresponding distinct implementation techniques for high performance. The classes and corresponding implementation techniques are:

- **Small**. Small messages can both be stored in registers and be copied without seriously impacting performance in either case. Current technology puts an upper bound on their size at about 32 bytes on a 32 bit processor with 32 registers. Implementation techniques for this class include carrying a message in registers and allowing the message to be copied to avoid coupling application and message transport interfaces. Hardware DMA functionality may not be useful for this class of communication because the messages are too short to amortize DMA setup costs.
- **Medium.** Medium messages are too large to fit in registers or to be copied without impacting performance, but too small to amortize the setup cost of a throughput-oriented transfer protocol. The result is a size range from around 40 bytes to a kilobyte or small number of kilobytes. Direct access to application memory and the use of DMA support, if available, are important to achieving performance.

- **Large**. The lower bound of this class of messages is the size at which the primary performance concern shifts from message latency or message throughput to data throughput within a single message. Optimizing for data throughput requires a protocol that maximizes use of the available bandwidth without causing receiver overrun. This necessitates transfers directly to and from application memory, and may require flow control. The resulting protocols often have a setup phase to validate the memory addresses on the remote system and initialize any required flow control. Messages using such a protocol must be large enough to amortize this setup overhead. This minimum size may be a kilobyte or more.

With the exception of Express Messages [9], most other messaging systems have recognized only the small and large message classes. FLIPC is designed to optimize performance of the medium class of messages by using a shared memory interface to avoid copying, an optimistic basic transfer protocol designed to move this class of messages, and a flow control architecture that does not involve the basic transport protocol.

4 Architecture and Design

FLIPC contains three major components, as shown in Figure 1:

- The *messaging engine*. This is the body of hardware and software that moves messages between nodes.
- A fixed size non-pageable *communication buffer* that is shared between the messaging engine and all applications that use FLIPC.

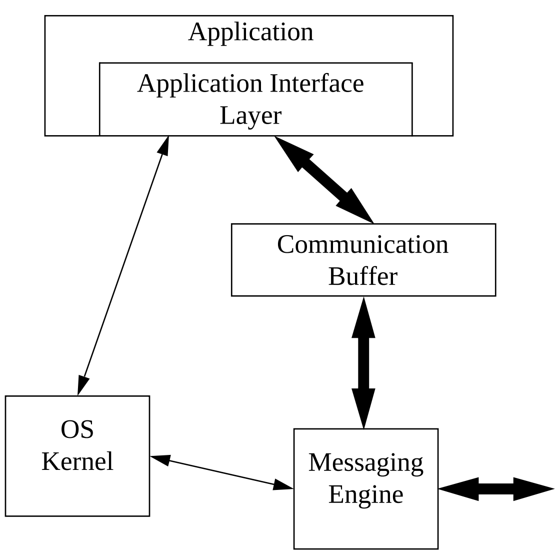

Figure 1: FLIPC Architecture

- An *application interface layer* that provides formal interfaces to applications and hides the data structures in the communication buffer. This consists of both a library and header file(s).

The *operating system kernel* is involved only in synchronization actions that cannot be directly accomplished via state in the communication buffer. The heavy arrows in Figure 1 indicate message flow, and the light arrows indicate synchronization operations that require interaction with the operating system kernel.

The messaging engine is an independently executing component of the system. It is intended to execute on the programmable controller in the communication interface when one is present, but can also be implemented as part of the operating system kernel for debugging purposes or on systems lacking the required hardware. Synchronization between the messaging engine and the application consists entirely of wait- free synchronization, making it impossible for an errant application to stall the communication controller.

The communication buffer is the focal point of FLIPC. It is located in shared memory accessible to both the application(s) and the messaging engine, and it contains all of the memory resources used for messaging. This design enables direct interactions between the application and the messaging engine without requiring code from either component to cross the protection boundary into the other. An additional result is that the OS Kernel is removed from the messaging path, making that path significantly shorter. Protection of the messaging engine from the application can be enforced via appropriate checks in the messaging engine, but can be removed to increase performance of a trusted application.

FLIPC implements multiple endpoints per communication buffer to support application requirements. Using different endpoints for different threads avoids contention for communication resources in multithreaded applications. Multiple cooperating applications can run on a single node by dividing or otherwise sharing the endpoints in a single communication buffer. The implementation of resource control at the endpoint level makes it easy to separate resources for different classes of traffic by using different endpoints.

An endpoint group logically combines multiple endpoints into a single abstraction. FLIPC supports a receive operation that retrieves a message from an endpoint if there is an available message on any endpoint in the group. This operation is implemented entirely in the library because the resource control model's association of buffers with endpoints makes it infeasible to merge the endpoint buffer queues. FLIPC also supports a blocking receive operation on an endpoint group. This design approach was chosen over the interrupting upcall methodology of active messages because application level interrupts are not appropriate for a multitasking real time environment. Interrupts disrupt execution in a way that cannot be controlled by the scheduler, reducing the real time predictability of the system. In contrast, FLIPC provides a real time semaphore option that causes the thread awakened by a message arrival to be presented to the scheduler in the OS kernel, allowing it to determine when it is appropriate to execute that thread.

Fixed size messages simplify the messaging engine logic. The actual size restriction is platform dependent. For the Intel Paragon, the characteristics of the DMA support in the interconnect interface require a message size that is at least 64 bytes and a multiple of 32 bytes. FLIPC uses 8 bytes of each message for internal addressing and synchronization purposes, so 56 bytes is the minimum application message size. Transfer of messages larger than the fixed size selected at boot time is not supported. FLIPC shields applications from buffer alignment restrictions by internalizing all message buffers. An application must call FLIPC to allocate a message buffer, allowing the implementation to ensure that all such buffers are correctly aligned.

Addressing facilities were designed to minimize implementation complexity. FLIPC message destinations (receive endpoint addresses) are opaque and determined by the system. This requires receivers to obtain endpoint addresses of endpoints they have allocated from FLIPC and pass those addresses to senders. FLIPC does not contain a nameservice of its own, but assumes that one is available for this purpose.

5 Message Transfer

FLIPC's basic messaging operation performs an asynchronous one way delivery of a fixed size message from a source endpoint to a destination endpoint. The send and receive interactions with the messaging engine are symmetric in that both involve queueing a message buffer for the messaging engine and subsequently determining whether the send or receive operation has completed. Both polling and blocking versions of completion detection are supported.

Figure 2 shows the complete sequence of operations involved in a message transfer. Step 1 involves the receiver providing a message buffer to receive the message. The sender sends the message at step 2 by queueing the message buffer on the endpoint for the messaging engine. The messaging engine transfers the message in step 3, queueing it on the receive endpoint by placing it in the buffer supplied at step 1. The receiver receives the message by removing it from the receive endpoint at step 4, and the sender recovers its buffer for reuse by removing it from the send endpoint at step 5. Steps 2 through 4 are the actual message delivery path, with steps 1 and 5 being associated resource control operations.

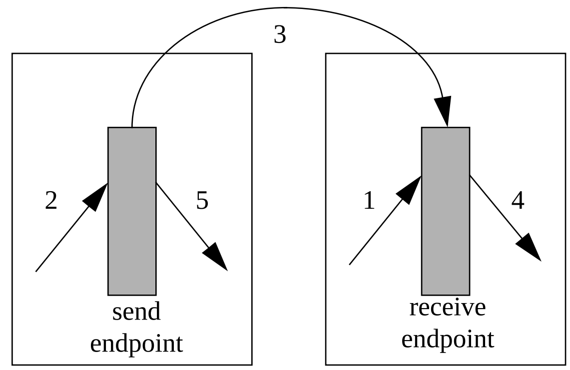

Figure 2: FLIPC Message Transfer

FLIPC uses a message delivery model that combines a reliable ordered transport with higher level responsibility for resource management and any associated flow control. The messaging engine provides a reliable transport that preserves order for messages sent from the same source endpoint to the same destination endpoint. Vesting responsibility for resource management and flow control in the application and the application library allows the transport to ensure that every node is always prepared to receive from the interconnect. If a receive occurs without an available buffer on the destination endpoint, the received message is discarded. The resulting guarantee that all messages in transit can always be accepted by their receiving node avoids deadlocks on reliable interconnects. FLIPC maintains a counter in each endpoint to track discarded message events, and makes this available to applications.

Flow control to avoid discarded messages can be provided either by applications or by libraries designed to fit between applications and FLIPC. This structure greatly simplifies the buffer management logic in FLIPC and allows flow control policies to be customized to application needs. In some cases, static properties of the application structure may remove the need for runtime flow control. One example is that an RPC interaction structure with a fixed set of clients can statically determine the number of buffers needed based on the maximum number of clients. Another example is that an application made up of strictly periodic components can often determine its worst case buffering needs in advance based on the maximum number of messages sent per time period.

6 Wait-Free Synchronization

Properties of the programmable controllers that FLIPC exploits require the use of wait-free techniques for synchronization between the applications and the messaging engine. This is necessitated by the fact that the controller on which the messaging engine executes may be the only means of communication with the rest of the system. Hence, a controller hang may render the node useless. Wait-free synchronization ensures that an ill-behaved application cannot stall the controller and cause damage in this fashion.

This synchronization problem is further complicated by the inability to assume that the programmable controller can execute atomic memory operations other than loads and stores. The resulting memory model is the same as that assumed by mutual exclusion algorithms such as Peterson's algorithm [15]. We simplified these synchronization problems by making the combination of the application and the application library component of FLIPC responsible for all synchronization among application threads. This reduces the wait-free synchronization problems to two-way problems between the messaging engine and an application thread. We solved these by separating or duplicating data structure components so that the application and messaging engine never attempt to concurrently write the same memory location. The synchronization among application threads is implemented via conventional multithreaded locking techniques based on a test and set lock, because these threads cannot execute on the programmable controller.

This type of wait-free synchronization requires a different approach to designing data structures because the only available atomic operations are reads and writes. For example, consider the counter used to record the number of discarded messages for each endpoint. FLIPC allows the application to read and reset this counter as a single logical operation, and guarantees that the reset will not cause dropped message events to be lost. A single memory location is not sufficient to implement a wait-free version of this counter because message drops occurring between the read and the write that resets the counter to zero will be lost. Instead FLIPC uses two memory locations to implement the counter. The first location is incremented each time a message is discarded, and the second location holds the value of this count as of the last 'read and reset' operation. The actual count of discarded messages is then the difference between these two values, and the reset is implemented by copying the value from the first location to the second location. This cleanly separates the data structure into one location written only by the messaging engine, the counter, and one written only by the application, the copied value.

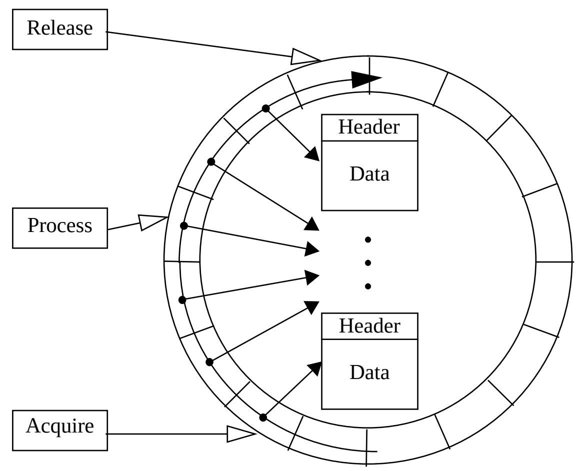

Figure 3: Endpoint Buffer Queue

The most important data structure for synchronization between the application and the messaging engine is the buffer queue in each endpoint. As shown in Figure 3, this structure consists of a circular queue of buffer pointers that tracks not only the queue head and tail, but also how far the messaging engine has proceeded through the queue. The solid circular arrow in the figure indicates the direction in which the release (head), process (middle), and acquire (tail) pointers move as buffers move through the endpoint. The application releases buffers to the endpoint by inserting pointers to them at the front of this queue. The messaging engine uses the process pointer to follow behind the front of the queue, sending from or receiving into these buffers depending on the endpoint type. Buffers processed by the messaging engine become free to be acquired by the application. This buffer acquisition occurs at the tail of the queue. The queue is empty when all three pointers point to the same location; the two half empty conditions of no buffers to process and no buffers to acquire occur when the corresponding two pointers point to the same cell. Each buffer also contains a state field that is changed when processing has been completed, allowing an application to determine when processing of a specific buffer is complete.

7 Implementation

FLIPC has been implemented on three different hardware platforms. The application interface library and communication buffer data structures are the same in all three cases, demonstrating the generality and portability of the system. On the other hand, the messaging engine is necessarily specific to the communication hardware and requires reimplementation for each platform. Our development platforms have been PC clusters interconnected by ethernet or a SCSI bus [3]. We have also implemented FLIPC on the Intel Paragon using the Paragon mesh interconnect. Our initial development employed transports for all three platforms developed by another project as OS kernel components to a common interface, the Kernel to Kernel Transport (KKT) interface [13]. This interface is not a good match to the one way messages used by FLIPC because KKT uses an RPC to deliver each message. On the other hand, this was very effective for development purposes, because it allowed us to focus on the platform independent components of FLIPC. The resulting development strategy allowed us to build and debug a fully functional system without using scarce Paragon time. When Paragon time became available, we were able to move the KKT implementation of FLIPC from the SCSI and ethernet clusters to the Paragon in less than a week including test time.

We then developed a native FLIPC messaging engine optimized and customized for the Paragon hardware, specifically Paragons that use MP3 nodes. These nodes contain three 50MHz i860 processors, one of which is reserved as a message coprocessor for accelerating messaging operations. Cache coherency is implemented among all three processors. This messaging engine executes directly on the Paragon's message coprocessor and accesses the communication buffer directly instead of using the KKT transport interface. The inter-node protocol for message transfer is an optimistic protocol that aggressively sends messages without acknowledgment or feedback. As noted previously, the receiving node discards messages if receive buffers are not available. This protocol coexists with other protocols in the Paragon's protocol framework on the message coprocessor, allowing multiple protocols to be used simultaneously[3]. For instance, our implementation of FLIPC on the OSF/1 AD operating system [22] requires both the FLIPC and OSF/1 AD protocols to operate simultaneously. Section 8 reports performance results for this implementation.

Tuning this implementation uncovered two performance problems caused by the cache coherency implementation on the multiprocessor Paragon nodes. The first problem was that the caches do not implement cache residency for multiprocessor locks. Instead, the test and set operation that acquires a lock must lock the bus and perform the read and write operations directly on memory, with a severe impact on performance. We avoided this problem by implementing versions of the message send and receive operations that do not use multiprocessor locks for mutual exclusion among application threads. These operations are intended for use by applications whose structure ensures that at most one thread will access each endpoint, or for which mutual exclusion can be provided at a higher level. All of our performance results use these interface versions. The second problem was that false sharing of variables written by both the messaging engine and the application library in the same 32 byte cache line on the Paragon caused excessive numbers of cache invalidations. We solved this problem by extending our design approach to wait-free synchronization to ensure that concurrent writes from the application and messaging engine can never occur in the same cache line. This reorganization of communication buffer data structures eliminated the false sharing. The combination of these two optimizations improved latency by 15µs or almost a factor of two. We had expected to encounter cache effects in tuning FLIPC, but were surprised by the magnitude of their impact.

8 Performance

Figure 4 shows the message latency for the optimized FLIPC implementation on the Paragon. The measured message latencies range from about 15.5µs to 17µs.

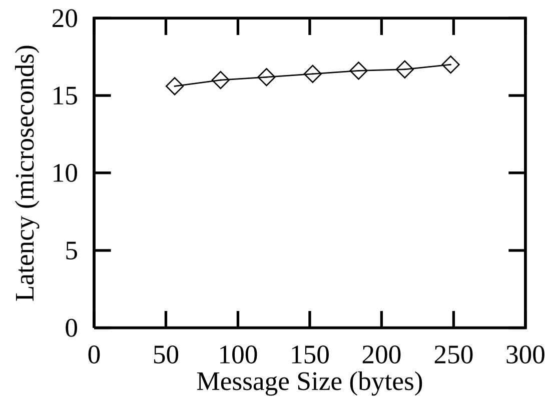

Figure 4: FLIPC Message Latency

3. The overall structure that makes it possible to integrate the FLIPC protocol into the Paragon's protocol framework is based on work originally done by Lok Liu as part of the Paragon Active Messages project [2].

The corresponding standard deviations range from 0.5μs to 0.65μs and are approximately the size of the symbols used in the figure. For message sizes of 96 bytes and above, the latencies obey the following equation:

Latency = 15.45μs + 6.25ns/byte

Shorter messages can be sent slightly faster due to changes in hardware behavior. The slope of 6.25 ns/byte indicates that increasing the FLIPC message size increases the use of interconnect bandwidth at over 150 MB/s. This is on an interconnect whose hardware peak bandwidth is 200 MB/s, and for which the best throughput achieved by any software is 160 MB/s [12].

These measurements were obtained via a test program that measures the time consumed by multiple two-way message exchanges between a pair of nodes. The time for a single message is then obtained by dividing this overall time by twice the number of two-way exchanges. These results are from a configuration that does not contain all of the validity checks that protect the messaging engine against corruption of the communication buffer by an errant or malicious application. Configuring these checks adds an additional 2μs to the above times.

Running the test program for a small number of exchanges yields results that are about 3μs faster than the above steady state results from test runs that include hundreds of message exchanges. We believe that the faster results for small numbers of exchanges are explained by cache start-up transients. The 16 kilobyte caches used by the i860 processors on the Paragon are relatively small, and the Paragon does not implement any secondary caches. The result is that the caches are sensitive to disturbances caused by executing code outside the inner test loop, so that saving the results of the previous test cycle is sufficient to cause replacement of a significant portion of the cache. The result of these replacements is that cache lines that are shared in the steady state of the test are not shared at start-up, resulting in fewer cache invalidations. This performance anomaly reinforces the importance of cache effects as an impact on performance of this class of messaging systems.

9 Related Work

FLIPC is most closely related to two systems, Paragon Active Messages (PAM) on the Paragon and Express Messages on the iPSC/2 hypercube.

PAM [2] consists of two communication subsystems, an active messages facility that transfers 20 byte messages, and a bulk transport facility based on direct read and write operations to remote memory. The active messages facility is layered on top of an optimistic transport protocol for fixed size messages via polling for incoming messages. PAM's optimizations for small messages and the simpler functionality by comparison to FLIPC yield a message latency of less than 10μs, about a third faster than FLIPC would be on a 20 byte message. An important difference caused by optimizing for small messages is that application preallocation and management of message buffers is not needed by PAM because a 20 byte message can be copied to or from an internal data structure at almost zero cost, less than 0.2μs. Functionality differences include the support for multiple endpoints, data streams, and the related resource control that is found in FLIPC, but not PAM. PAM's bulk transport mechanism is complementary to FLIPC, as FLIPC does not contain a bulk transport mechanism.

There are strong similarities between the PAM and FLIPC implementations in that both use a wired communication buffer shared between the application and messaging engine, and an optimistic transport that discards messages when receive resources are not available. Both systems recognize that a dedicated buffer avoids deadlock problems that may arise in the use of an optimistic transport on a reliable interconnect, although the actual buffer designs are quite different. Both systems also move flow control support from the basic internode protocol to higher layers to support customizing flow control to application requirements. The window based flow control protocol used by PAM's active message facility is an example of this customization. PAM is targeted to smaller messages than FLIPC; PAM messages are 28 bytes, of which 8 bytes are used by PAM, leaving 20 for the application. PAM implements active messages on top of its transport by using 4 bytes of the message to hold the address of the remote message handler; the same implementation could be used on FLIPC.

Express Messages [9] was a messaging facility for the iPSC/2 hypercube. This system contained several ideas that were utilized and enhanced in the design of FLIPC. Express Messages recognized the distinction among small, medium, and large messages, and also used an aggressive optimistic transfer protocol for medium messages. Fixed size message buffers were used for medium messages, but via page mapping techniques instead of a shared memory buffer. A shared control bit was used switch between polling and interrupt-based message delivery mechanisms, but system calls were used for buffer management in contrast to the shared data structure implementation in FLIPC. The absence of kernel threading support in the NX/2 operating system required Express Messages to implement threading at user level, including a handoff mechanism that executed an interrupted critical section if the interrupt handler needed to enter the section. FLIPC is based on kernel thread support, allowing it to deliver messages to threads rather than upcall handlers. This avoids interrupting critical sections. Instead, the message that would have caused an interruption is handled by a thread, allowing conventional multithreaded synchronization techniques to be used to resolve critical section conflicts.

Two other common high performance messaging systems for the Paragon are NX [11] and SUNMOS [21]. Both optimize performance of large messages for high performance numerical computation, and SUNMOS also optimizes zero length messages. NX is part of the basic Paragon operating system, whereas SUNMOS is a single application operating system that is run on a subset of the Paragon nodes, utilizing the Paragon operating system running on the remainder of the machine for the services it does not provide. NX achieves a bandwidth[4] of over 140 MB/sec and SUNMOS approaches 160 MB/s for sufficiently large messages. Aside from the underlying hardware architecture, FLIPC has little in common with these systems because they are optimized for large messages rather than medium messages. FLIPC is complementary to such systems, as a bulk transfer mechanism needs to be added to FLIPC to obtain a complete system. SUNMOS has the further complication that its basic messaging protocol assumes a non-multiprogrammed system by sending multi-megabyte messages as a single packet. This occupies the path through the interconnect for the duration of the message and is a potential responsiveness problem in a real time environment.

The optimized FLIPC implementation for the Paragon significantly exceeds the performance of these other messaging systems on medium sized messages. The FLIPC latency for a 120 byte message transferred between applications on two Paragon nodes is 16.2µs. The comparable latencies for the three other messaging systems on the Paragon are:

- NX (Paragon O/S R1.3.2), 46µs [11].
- Paragon Active Messages, 26µs [2].
- SUNMOS, 28µs [12][21].

This demonstrates the performance impact of not optimizing for the medium class of messages. These three messaging systems have been optimized for bandwidth on large messages. In addition, PAM has been optimized for latency on very small messages.

10 Future Work

There are a number of opportunities for future work to improve on FLIPC. The explicit resource management model is effective, but is also awkward to program. Our experience is that a FLIPC application can expect to employ about half of its calls to FLIPC to send or receive messages, and the other half for message buffer management. An improved buffer management design that frees the programmer from most of these details is clearly called for. Support for multiple communication buffers per node and protection mechanisms that restrict where messages can be sent should be added to support multiple applications that do not trust each other. FLIPC was designed solely to address the transport of medium sized messages and needs to be integrated into a system that provides excellent performance for messages of all sizes. As part of this work, we are considering extensions that allow applications to indirectly access memory on other nodes [16]; some related ideas can be found in the SUNMOS [21], PAM [2], and Illinois Fast Messages [7] systems. Finally, we intend to pursue further integration of FLIPC into a real time environment by adding real time prioritization and capacity/bandwidth control functionality to the basic inter-node transport.

11 Conclusion

FLIPC incorporates an innovative combination of design and implementation techniques that achieve excellent performance for medium messages on high speed communication hardware in a real time environment. The most important architectural approach is to recognize and explicitly target programmable communication controllers. The FLIPC design manifests this targeting in several ways:

- Logical separation of messaging engine and OS kernel allows the messaging engine to be implemented on the communication controller.
- Asynchronous messaging interfaces and wait-free synchronization decouple the communication controller from applications.
- The shared communication buffer allows the OS kernel to be bypassed.

4. The NX performance results reported in [2] are from an older release of NX. This result is from a more recent version (1.3.2).

FLIPC is designed for medium messages, an important class of messages for distributed real time applications. The increased performance it obtains by comparison to other messaging systems indicates that this size class of messages has been overlooked by these other systems.

Our experiences in implementing FLIPC provides some useful guidance for designers of other systems. The importance of cache management to the performance of distributed messaging systems is indicated by the cache problems we encountered during performance tuning, and the factor of two in performance obtained by tuning for cache effects. The limitations of some programmable communication controllers required us to address wait-free synchronization problems in a memory model without atomic read-modify-write operations. Finally, employing PC clusters proved to be an effective way to develop prototypes, including validating the portability of FLIPC across two different communication mediums. This greatly reduced our need for Paragon time to develop the final version of FLIPC, because the platform independent components had already been implemented and debugged.

Acknowledgments

For their assistance in making this work possible, we are grateful to the Intel Scalable Systems Division, especially Paul Pierce and Roy Larsen. We would also like to thank the Active Messages group of the University of California at Berkeley, and especially Lok Liu for making the source code of Paragon Active Messages available for us to study. Paul Davis of Honeywell kindly provided measurements on the current version of NX. Dejan Milojicic and the anonymous Usenix reviewers provided comments that greatly improved this paper.

References

[1] 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*, February 1995.

[2] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz, 'Remote Queues: Exposing Messages for Optimization and Atomicity', *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1995, pp. 42-53.

[3] Randall W. Dean, Michelle M. Dominijanni, Steven J. Sears, and Alan Langerman, 'SCSI for Host to Host Communication', OSF Research Institute Technical Report, In preparation.

[4] Peter Druschel, Larry L. Peterson, and Bruce S. Davie, 'Experiences with a High-Speed Network Adapter: A Software Perspective', *1994 SIGCOMM Conference Proceedings*, October 1994, pp. 2-13.

[5] 'High Performance Distributed Computing (Hiper-D) Integrated Demonstration One Report', Naval Surface Warfare Center (NSWC), Dahlgren, Virginia, September 1994.

[6] Intel Corporation, *Intel Paragon XP/S Supercomputer Spec Sheet,* 1992.

[7] Vijay Karamcheti and Andrew A. Chien, 'A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D', *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 298-307.

[8] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy, 'The Stanford FLASH Multiprocessor', *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 302-313.

[9] J. William Lee, 'Performance of User-Level Communication on Distributed Memory Multiprocessors with and Optimistic Protocol', Technical Report 93-12-06, Department of Computer Science and Engineering, University of Washington, December 1993.

[10] NCR Corporation, *NCR 53C825 PCI-SCSI I/O Processor With Local ROM Interface Data Manual*, 1993.

[11] Paul Pierce and Greg Regnier, 'The Paragon Implementation of the NX Message Passing Interface', Technical Report, Intel Scalable Systems Division, Beaverton, Oregon.

[12] Rolf Riesen, 'SUNMOS Performance', Sandia National Laboratories web page linked to http://www.cs.sandia.gov/~rolf/puma/sunmos/sunmos.html.

[13] Steve Sears, Michelle Dominijanni, Alan Langerman, and David Black, 'Kernel to Kernel Transport Interface for the Mach Kernel', Technical Report in OSF Research Institute Collected Papers, Operating Systems Volume 3, April 1994.

[14] Daniel P. Siewiorek, C. Gordon Bell and Allen Newell, *Computer Structures: Principles and Examples*, New York: McGraw Hill Book Company, 1982.

[15] Andrew S. Tannenbaum, *Modern Operating Systems,* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1992, page 37.

[16] Chandramohan A. Thekkath, Henry M. Levy, and Edward P. Laxowska, 'Separating Data and Control Transfer in Distributed Operating Systems', *Proceedings or the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, *Operating Systems Review,* vol. 28, num. 4, December 1994, pp. 2-11.

[17] Lewis W. Tucker and Alan Mainwaring, 'CMMD: Active Messages on the CM-5', *Parallel Computing*, vol. 20, 1994, pp. 481-496.

[18] Ronald J. Vetter, ‘ATM Concepts, Architectures, and Protocols’, *Communications of the ACM*, vol. 38, num. 2, February 1995, pp. 30-38.

[19] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, ‘Active Messages: a Mechanism for Integrated Communication and Computation’, *Proceedings of the 19th International Conference on Computer Architecture*, May 1992, pp. 256-267.

[20] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham, ‘Efficient Software-Based Fault Isolation’, *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993, pp. 203-216.

[21] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup, ‘PUMA: An Operating System for Massively Parallel Systems’, *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, January 1994, pp. 56-65.

[22] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, and Durriya Netterwala, ‘An OSF/1 Unix for Massively Parallel Multicomputers’, *Proceedings of the Winter 1993 USENIX Technical Conference*, January 1993, pp. 449-468.

Author Information

David L. Black is a Research Scientist at the OSF Research institute, where he has been working on operating systems of various sorts since 1990. Previously, he was a graduate student at Carnegie Mellon University performing research on Mach, and learning how long it takes to write a Ph.D. thesis. His research interests include operating systems, multiprocessors, and real-time computing.

Randall D. Smith has been working on high-speed application messaging systems for three years, at Thinking Machines Corporation and the OSF Research Institute. He also spent three years in graduate study at the Massachusetts Institute of Technology in Computational Neuroscience, and a year working for the Free Software Foundation hacking GDB and GLD.

Steven J. Sears is currently a Senior Consulting Software Engineer with Novell Corporate Research and Development, attached to the Advanced Systems Architecture & Technologies Group. Before joining Novell he solved ‘hard problems’ at the OSF Research Institute. His interests include distributed operating systems and software, computer architecture, high speed networking, and distributed memory systems.

Randall W. Dean is a Senior Research Engineer at the OSF Research Institute where he is currently working on distributed operating system issues. Previous to joining the OSF in 1993, he spent eight years on the research staff at Carnegie Mellon University, the last four on the Mach project. His research interests include operating systems, multiprocessors, distributed computing and distributed memory management.