Optimized Memory-Based Messaging: Leveraging the Memory System for High-Performance Communication

David R. Cheriton and Robert A. Kutter
Stanford University

ABSTRACT: *Memory-based messaging*, passing messages between programs through a shared memory segment, is a recognized technique for efficient communication that takes direct advantage of memory system performance. However, the conventional operating system and hardware support for this approach is inefficient, especially in large-scale multiprocessor systems.

This paper describes interface, software and hardware optimizations for memory-based messaging that efficiently exploit the basic mechanisms of the memory system to provide superior communication performance. We describe the overall model of optimized memory-based messaging, its implementation in an operating system kernel and hardware support for this approach in a scalable multiprocessor architecture. The optimizations include address-valued signals, message-oriented memory consistency and automatic signaling on write. Performance evaluations show these extensions provide a three-to-five-fold improvement in communication performance over a comparable software-only implementation.

1. Introduction

The performance of operating system communication facilities significantly influences the performance and modularity of the system and its applications. In essence, the slower the communication, the slower the applications that rely significantly on communications. Moreover, slow communication increases the cost of modularity, leading to less modular systems and applications. Despite increasing processor and memory system performance, several trends suggest continuing importance to interprocess communication performance.

First, parallel applications require high-performance communication to exploit relatively fine grain tasking to achieve a high degree of parallelism. For example, many parallel programs are structured in a work queue model in which threads allocate work from a shared work queue. It is more efficient in terms of memory traffic for the processors to communicate by messages with a processor running an allocator than to trap the code and data associated with the allocator into its cache and then execute the allocator itself, incurring the delay and memory overload of shared memory consistency. (The requesting processor is unlikely to have been the last to execute this allocator.) Fast messaging minimizes the overhead for this work allocation.

Second, many new applications require high input/output performance, placing demands on communication system facilities. For example, moving video from a network interface to a multimedia application and then onto a display requires more communication bandwidth than conventional approaches have been designed to provide. With gigabit networks, direct video input, disk striping [Patterson et al. 1988], cylinder caching, and solid-state disks improving the I/O device performance, the internal communication system can become a significant overhead.

Third, in a system structured as a micro-kernel with protected user-level servers, an efficient communication system allows access to system services and implementation of those system services without a significant performance penalty. For example, an application file open operation may access a directory service, a file server, and a caching service, thus incurring the cost of several protected inter-address space communications rather than a single kernel trap as with a conventional monolithic kernel. The modular and protected structure of the

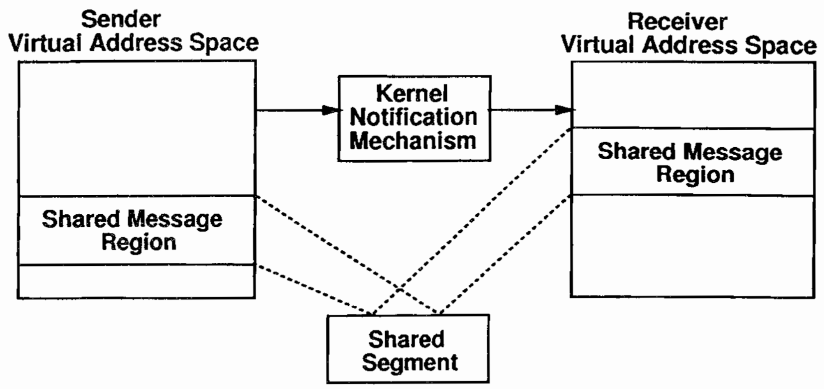

Figure 1. Two processes communicating through shared memory.

micro-kernel approach seems particularly beneficial for large-scale systems, where it is not acceptable to have a single error in "the operating system" bring down the whole machine.

Most of the work on IPC and RPC systems has focused on the *copy* model of communication. The data to be communicated is passed to a message-passing facility that logically copies the data to the recipient(s). The copy model provides safe and simple semantics. However, the cost of data copying is significant in many systems [Fitzgerald 1986; Hansen 1973; Schroeder & Burrows 1989], especially for larger data units and with caching architectures. Copying produces poor cache behavior [Torrellas et al. 1990] with a performance penalty that increases with the increasing ratio of processor speed to average memory access time. This cost can be reduced, to some degree, by providing virtual memory system support to remap data, rather than copy it, and by providing a copy-on-write mechanism to preserve copy semantics (e.g., Accent and Mach [Accetta et al. 1986]). However, the cost of remapping data in multiprocessor systems, as an alternative to copying, is greater than in uniprocessors [Black et al. 1989; Rosenburg 1989] because of the need to update or invalidate the TLB or page table in each processor. This optimization also forces communication into page-sized units.

As an alternative to this copy model, *memory-based messaging* uses a shared memory communication area between processes, as illustrated in Figure 1. A shared memory segment is created to act as a communication channel, the source and destination processes bind this segment into their respective address spaces, messages are written to this segment, and messages are read from this segment after some form of notification at the destination(s). Because the sending and receiving processes have direct access to the shared memory segment, the number of copies is reduced typically by a factor of two over the copy model primitives. For example, in an RPC system, the caller can marshal its call parameters directly into the shared memory segment and the callee can marshal these parameters directly out of the shared memory segment, eliminating the extra copies both into and out of a message buffer that arises with the copy model.

This approach has been used by a variety of commercial applications using the shared memory mapping facilities in Unix System V [Bach 1986].[1] It has also been used in some research systems, including the Berkeley DASH project [Tzou & Anderson 1991] and URPC [Bershad 1990].

Nevertheless, the performance of memory-based messaging is an issue for several reasons.First, notifying receivers of message arrival can be expensive because of the need to lock, queue and interrupt at the receiving processor(s). Second, the cost of accessing data in a shared memory between processors increases with the size of the memory system particularly because of cost of maintaining coherency in larger-scale shared memory systems. These overheads are significant even in small-scale parallel systems because of the increased ratio of processor speed to memory access time.

This paper describes *optimized memory-based messaging* which extends the basic memory-based messaging model with three key optimizations. We present these optimizations, their implementation in a combination of software and hardware in an operating system kernel and multiprocessor hardware system we have developed, and measurements of the performance of this system. We also include the results of analysis and simulation to predict the benefits of this approach for future machines, which are expected to have larger numbers of faster processors, larger memory systems and faster interconnection mechanisms.

The next section presents the memory-based messaging optimizations and the relevant details of their software and hardware implementationin an extended version of the V distributed System [Cheriton 1988.7] and the ParaDiGM multiprocessor [Cheriton 1991]. Section 3 describes our RPC implementation. Section 4 presents our measurements of this configuration. Section 5 describes the results of our analysis and simulation to determine the benefits of optimized memory-based messaging on some possible future computer systems. Section 6 describes previous research we see as relevant to this work. We close with a summary of the work, our conclusions, and some indication of future work.

1. Unix is a trademark of Unix System Laboratories.

2. *Optimized Memory-Based Messaging*

Optimized memory-based messaging incorporates with three key optimizations over the basic mechanism shown in Figure 1 namely:

- Address-valued signals
- Message-oriented memory consistency
- Automatic signal-on-write

Address-valued signals provide efficient, low-latency notification of message reception at the receiver(s). Message-oriented memory consistency reduces the transmission cost of message data through the memory system from sender to receiver(s). Automatic signal-on-write minimizes the sender cost of generating the signal. The following subsections describe these refinements and their implementation in detail. We show that these optimizations provide a simple and fast implementation, especially for scalable multiprocessor architectures.

2.1. *Address-Valued Signaling*

An *address-valued signal* is a signal that transmits a single virtual address from the signaling thread to one or more receiving threads, delivering the signal to a signal handler function with a single parameter, the address value. This facility contrasts to Unix signals and typical hardware interrupts, which do not allow a value to be transmitted. It also contrasts with conventional messaging that supports large, variable-sized data transfer with attendant complexity and performance costs.

The transmitted address value is translated before delivery from the virtual address provided by the signaling thread to the corresponding virtual address in the address space of the receiving thread, as illustrated in Figure 2. As shown in this figure, a signal specifying a virtual address in the shared memory region[2] is mapped to the corresponding offset in the shared segment. The signal is then delivered to each receiver with the virtual address mapping to this offset in the shared segment. More simply defined, the virtual address delivered to a receiver points to the same location in the shared segment (as mapped into the receiver's address space) as the virtual address specified by the signaling thread does in its address space.

2. A *region* refers to a range of a virtual address space.

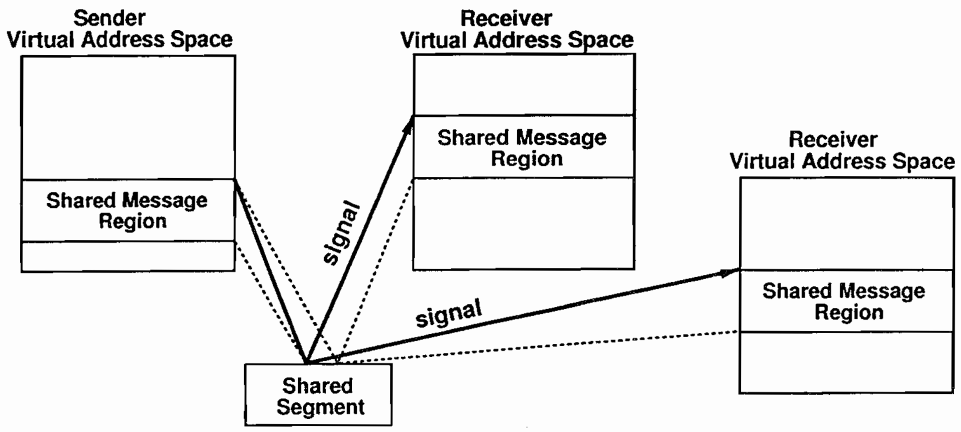

Figure 2. Address-valued Signaling for Notification.

In the current and expected use of memory-based messaging, a thread sending a message writes the message data into a free area of the message region associated with the destination thread(s) and then signals using the virtual address of this free area. The signal handler in each recipient is called with the (translated) address of this message, and the signal handler uses this address to access the new message and deliver it to the application. Protocols and conventions required between processes to set up shared segments, manage the allocation and release of message areas in the shared segment and define the actual message representation are discussed further in Section 3.

2.1.1. Kernel Interface

The following kernel calls support address-valued signals in our system.

- `SignalHandler(char *vaddr, int vaddrSize, void (*sigFunction)(char *vaddr));` Specify the signal-handling procedure `sigFunction` on the address range specified by `vaddr` and `vaddrSize`.
- `Signal(char *vaddr);` Generate an address-valued signal for the specified address.
- `Time SigWait(Time t);` Delay a thread until a signal arrives or until the requested time interval *t* has passed. The amount of time remaining from the requested time is returned.

The signal procedure specified for a given region is executed by a designated thread associated with this enabled region. This is normally the thread that set the signal handler by calling `SignalHandler`. As in other systems such as Unix that employ software signal mechanisms, this thread can either wait explicitly for signals using `SigWait` or simply receive signals as asynchronous procedure calls during its execution. The timeout parameter for `SigWait` efficiently supports the common case of a thread waiting for a signal or a timeout period, whichever comes first.

With appropriate hardware support, described in Section 2.1.2, a signal can be generated directly by writing a memory location, normally as part of writing a message into the message segment. The `Signal` call is used in the absence of this hardware support. It is also used when a signal is to be sent without writing a message, given that the signal mechanism can be used as a general notification mechanism for shared data structures.

A thread can have signal handlers enabled on several different regions simultaneously. Multiple threads in the same address space can have separate signal handlers on the same region, with signals being delivered concurrently to their respective threads. Signals to the same thread are delivered in FIFO order, rather than stacking the signals to provide a LIFO ordering. To date, FIFO delivery and non-blocking synchronization techniques in our implementation have obviated the need for enabling and disabling signals during signal handling as is common practice in Unix.

Address-valued signals have no associated priority but are executed with the priority of the thread that executes the signal handler. Threads with different priorities can enable signal handlers on regions of memory so that each memory region represents a different signal priority. Supporting separate signal priorities would require a set of buffers, one for each priority, and a mechanism to communicate the signal's priority to the memory system hardware. Given the complex logic and the redundancy with thread priority, the cost of providing prioritized signaling appears unjustified.

2.1.2. Implementation

Address-valued signaling is implemented in software as an extension of conventional modern virtual memory facilities supporting mapped files and shared segments. In particular, a memory region descriptor records the signal function and thread for the signal handler on that region, if any. The mapping of the signal physical address to virtual address(es) and thread(s) uses the standard inverse page mapping data structures required in the virtual memory system. Our implementation also includes as an optimization a fast hash table that maps physical addresses to virtual addresses and signal handlers.

The kernel `Signal` operation first translates the virtual address to a physical address. In our implementation, it uses the fast virtual to physical mapping function supporting TLB fault handling required by the virtual memory mechanism.

It then attempts to map the physical address to a virtual address and signal handler using the fast hash table. If that fails, the physical address is then mapped to a page descriptor which identifies the segment, which then maps to zero or more regions each with a signal function and thread for the signaler handler, if non-null. (If there is a single signal thread for this address, it loads this mapping to the hash table if it is not already present.)

Delivery of the signal is similar to Unix signal delivery when the thread is not executing a signal handler. The kernel saves the thread context, creates a stack frame to call the signal handler function associated with the signaled memory region, and passes the translated signal address as the parameter. On return from the signal handler, the thread resumes without reentering the kernel. Trapping to the kernel after the signal handler finishes is avoided by storing the thread context in user-accessible memory and setting up the thread stack to return to a run-time library function that restores the thread context. This optimization is compatible with all the processors supported by our system, including the MIPS R3000 and the Motorola 68040. If a thread is already executing a signal handler at the time of signal delivery, the signal is queued in a signal FIFO page associated with the thread or dropped if this area is full. The signal delivery code is similar to that used to implement our emulator signals [Cheriton et al. 1990] and exception-handling.

In a multiprocessor system, if some of the threads to be signaled are assigned to other processors, the signaling processor must interrupt these other processors and pass the physical address to each of these processors. A typical implementation entails a message queue structure per processor with single-word messages, synchronized for multiprocessor access, plus a hardware interprocessor interrupt facility. This implementation is feasible in small-scale systems but has a significant cost in large-scale systems because of memory contention. A simple hardware extension of the conventional interrupt system can support address-valued signals efficiently, as described next.

2.1.3. Hardware Support

In our extended interrupt system, each processor has a FIFO that stores memory addresses representing signals delivered to this processor but not yet processed. The signal address is transmitted over each interprocessor bus as a special signal bus transaction including the physical signal address and control lines specifying the affected processors. When a processor receives a signal bus transaction and is one of the processors selected by the control lines, its bus interface stores the address in the processor's FIFO buffer and interrupts the processor. In our current implementation, each FIFO buffer has 128 entries so signal loss is very unlikely, but not impossible. When a processor receives the signal interrupt, it takes the next physical address from its FIFO buffer, translates it to each virtual address that maps to this physical address, and delivers the signal to each thread associated with this signal region that is assigned to this processor.

In our prototype implementation in the ParaDiGM system, the memory system determines the signaled processors from a (cache) directory indexed by the physical address as part of the automatic signal-on-write facility, as described in Section 2.3. However, a simple hardware interface would be a control register that the software could write with physical address and processor bitset that would generate the signal bus transaction.

Our implementation requires no modifications to the processor chip itself. However, in an extended implementation, the time to deliver a signal to a thread could be further reduced by using a reverse TLB integrated into a processor specifically designed to support memory-based messaging. The *reverse TLB (RTLB)* would provide translation from a physical address to a signal program counter (PC), virtual address, and priority. (The priority could be stored in the low-order bits of the virtual address to avoid having a separate field in the RTLB.) If a physical address did not match any of the RTLB entries, the RTLB would provide a fixed value for the signal PC and pass through the physical address as the virtual address value. On a signal interrupt, the processor would read the interrupt PC, virtual address, and priority from a reverse TLB. In the interrupt mode specified by the priority, it would then branch to the specified interrupt PC with the virtual address in a register. (One level of priority would designate user mode.) The operating system software would load the RTLB on each thread context switch when the new thread had signals defined.[3] Based on our experience to date, we believe that a 4–8 entry RTLB would perform well. Thus, with this extension, a signal handler would be called in user space with no operating system intervention, at least in the expected case. Therefore, the time from the point a signal is generated to the execution of the user-defined signal handler code would be less than 7 processor cycles.

The extended design appears feasible even for RISC processor design for several reasons. First, current RISC processors already perform similar actions on interrupts, exceptions, and resets: they read an address to branch to, set the interrupt priority, and set a cause register (as in the R4000 architecture). This extension simply requires these values to come from the RTLB, rather than

3. This mechanism would not handle the case of a signal that was enabled on more than one thread executing on the same processor. For this case, the standard kernel signal handler would be invoked, using the software mechanism described earlier.

some fixed interrupt vector. Second, the RTLB is small and could use a similar design to those used with standard on-chip TLBs. Thus, putting it on the processor chip is only a matter of chip real estate. If processor chip real estate is tight, the RTLB could be integrated with the FIFO mechanism at the cost of having to transfer more from the FIFO to the processor on an interrupt. Finally, this mechanism could replace the conventional interrupt and exception mechanism, especially if the RTLB is integrated in the processor chip. For example, with a R4000-like architecture revised along these lines, the cause register would be replaced by a "signal address register" and, on exception, the exception type could be encoded in well-known values placed in this register by the exception mechanism. This extension simply generalizes ad hoc techniques in the interrupt mechanisms of RISC processors and does not add significant additional control logic or any new registers. It appears like an attractive direction for processor chip design if the memory-based messaging approach of this paper become popular.

2.1.4. Advantages

Address-valued signaling provides a simple efficient notification mechanism for memory-based messaging. The translated signal address provides a direct, immediate, and asynchronous message specification to the recipient(s), allowing each recipient to immediately locate the message within the segment. In particular, the same signal handler procedure can be used for several different segments and still immediately locate the signaled message using the supplied virtual address. The application can also have different signal handlers bound to different memory regions. The particular signal handler is selected automatically based on the region of memory in which the signal occurs.

Address-valued signals can be used for other purposes than memory-based messaging. For example, a thread can notify other threads of a change to some object in a shared memory segment that the threads are all accessing. This change notification is a common facility in various object-oriented programming frameworks and can benefit from the operating system support in a multiple address space or multi-processor implementation.

In contrast to our scheme, conventional Unix signals provide no ability to pass such a parameter and thus require the recipient to either search or query for the message or be tied to some fixed convention on the location of the next message with the signal handler specially coded for each segment to known that location. For example, a familiar approach in Unix is to map all asynchronous I/O to the same signal (`SIGIO`) and then use a `select` operation, with the resulting extra system call and overhead, in the signal handler to determine the file or device on which to act.[4]

Address-valued signaling is also significantly more efficient than the Unix System V `msgsnd` and `msgrcv` system calls and conventional message primitives of the various message-based operating systems, as shown in Section 4.

Address-valued signaling allows an efficient, scalable hardware implementation providing minimal processor and memory system overhead in large-scale multiprocessor systems. The FIFO buffer eliminates the need for software delivery of the address values and for synchronized software queues to hold those addresses. Shared memory message queues and their associated locks cause significant memory system overhead in larger scale systems because of the potentially high memory contention on the queue data structures. In particular, two or more processors in widely separated portions of the memory system contending for a shared queue can produce thrashing of the cache line(s) holding the queue lock and data.

Finally, implementation of address-valued signaling is relatively simple, both in hardware and software. The implementation takes advantage of the conventional virtual memory mapping hardware and software data structures to deliver signals to the appropriate signal handler within the desired thread. Because address-valued signal delivery is integrated with the memory system, there is no need for a separate mapping, queuing, and protection mechanism, as arises with conventional message-based operating systems and the Unix System V message facilities.

The additional hardware to support address-valued signaling is a small percentage of the overall hardware cost (less than 1% in our implementation), and arguably close to zero cost in large configurations. In fact, the most significant hardware component, a per-processor FIFO buffer to store signal values, is required for interprocessor and device interrupts on large-scale systems in any case because the conventional dedicated bus line from interrupter to interruptee is not feasible. The FIFO buffer stores the interrupt, allowing the sending processor or device to send the interrupt across the interconnection network and not hold a connection.[5] Storing a single address, rather than the entire message, costs essentially the same as storing a potentially smaller value such as processor identifier or device identifier. We note that all device interrupts in our system are handled as address-valued signals, unifying and simplifying the hardware and OS software

4. Address-valued signals can easily subsume Unix signals by designating a unique set of virtual memory addresses that map to standard Unix signal numbers.
5. A separate synchronization bus has been used in small-scale multiprocessors, such as the SGI Power Series, but this approach appears to be even more expensive.

around this one general mechanism and avoiding the conventional *ad hoc* techniques used with devices.

2.2. Message-Oriented Memory Consistency

Message-oriented memory consistency is a consistency mode for a memory segment in which the reader of the segment is only guaranteed to see the last write to this memory after it has received an address-valued signal corresponding to that write. If a signal is lost, the receiver is not guaranteed to see the update at all. Moreover, the message can be overwritten by a subsequent message in the FIFO buffer before the receiver reads it.

These semantics match those of a network receive buffer. A thread can only expect a new packet to be available after an interrupt from the interface, not at the time it is written. Also, if a packet is not received, or is received in a corrupted form, or is overwritten, the data is not available at all. (Section 3 describes our techniques for detecting and recovering from these errors.)

2.2.1. Kernel Interface

Message-oriented memory consistency is specified as a property of a segment at the time of its creation using the `CreateSegment` system call in the extended V kernel.

- `Segment *CreateSegment(int attributes, int mode, int flags, int *error);` Create a segment that, assuming the `mode` parameter is set to `MESSAGE_CONSISTENCY`, uses message-oriented consistency. On a machine that provides hardware support for this model, this information is stored in the cache directory. The `flags` parameter can be set to `CACHE_LINE` or `PAGE` to indicate the unit on which to signal, and thus the unit of message consistency. If it is set to `CACHE_LINE`, an address-valued signal is generated when the sending thread writes the end of a cache line, otherwise at the end of a page.

The `CreateSegment` operation is roughly equivalent to the Unix System V `shmget` or the BSD Unix `open` system calls. These segments are also similar to BSD Unix mmap'ed open files.

Memory segments with message-oriented memory consistency are intended to be used unidirectionally as part of memory-based messaging. One thread binds the segment as writable and others bind it as read-only. Consequently, there is generally a single writer for a set of addresses within the segment. However, a shared channel may also have multiple writers, just like a CB radio channel. For example, clients may use a well-known channel to multicast to locate a server.

2.2.2. *Implementation*

The basic software implementation of message-oriented consistency is to tag the pages in a segment in `MESSAGE_CONSISTENCY` mode as non-coherent and use cache flush and invalidate instructions to flush the message data after writing it and invalidate cache lines at the point of reception of the signal. For example, the PowerPC [May et al. 1994] processor provides a page-granularity tag that can indicate whether memory coherency is required or not. It also provides unprivileged instructions for flushing and invalidating the processor cache on a cache line basis. Another example of this kind of instruction can be found on the MIPS R4000 processor. The CACHE control instruction [Heinrich 1993] can be executed from user mode to push a specified cache line from the first-level cache. Unfortunately, the MC68040 used in our prototype implementation only provides cache control instructions that operate in privileged mode using physical addresses. This shortcoming limits the performance because message area invalidation requires kernel involvement.

2.2.3. *Memory System Support*

Message-oriented consistency support in the memory system allows it to propagate the message data to the lower-levels of the memory system as well as other peer-level caches without incurring the overhead of conventional memory coherency protocol. (With no coherency at the L2 level and lower, a flush of message data might just force the data into the sending processor's associated second level cache and yet not make it visible to other portions of the memory system.)

In our ParaDiGM prototype, the memory system architecture uses the hierarchical cache structure shown in Figure 3 to implement a scalable shared memory. Each cache line unit has a cache directory entry containing a 3-bit mode and various other tag bits describing its state, as shown in Figure 4.

To efficiently support message-oriented consistency, the mode (MMM) field encodes a special *message* mode in addition to the conventional shared, private and invalid states of an ownership-based consistency protocol. The tag bits include a set of P_i bits, one per processor sharing the cache. These bits are required in the conventional *shared* state to indicate the processors with copies of the cache line. In message mode, the P_i bits indicate the caches that need to be informed when this cache line data is updated. The G or "global" bit indicates the next lower level of the memory hierarchy should be modified when this cache line is modified. The lower level maintains its own directory and further propagates the signal to other clusters of processors, as indicated by its tags.

When a segment is specified as in `MESSAGE_CONSISTENCY` mode, the software ensures that each cache line that is loaded with a cache block from a page in this

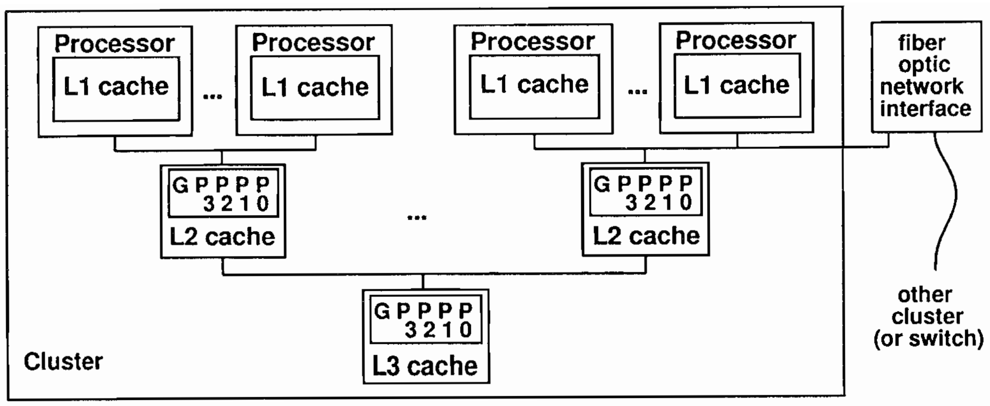

Figure 3. Multiprocessor Architecture.

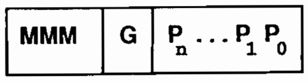

Figure 4. Hardware Cache Directory Entry.

segment is set in message mode. In our prototype, the second-level cache miss handler extracts this information from third-level cache tags which are set from page frame and segment information maintained by the operating system.

When a processor flushes L1 data back to a cache line in the second level cache that is in message mode, the cache controller then generates signals to all caches indicated by the P_i bits as (potential) recipients. If the G bit is set, the cache line is written through to the L3 cache which then similarly signals each L2 cache above it that is marked as a potential receiver by the P_i bits in the associated L3 cache tags. Each signaled cache either invalidates the corresponding cache line or else reloads this cache line if it is present in the signaled cache. Consequently, when a receiving processor invalidates this cache line in its L1 cache and then reads this cache line after receiving the signal, it is assured of reading the updated message data.

In our prototype, the message mode did not increase the cost per cache directory entry because there were extra code values available beyond those used by the conventional shared memory states. In the worst-case, it would require an extra bit per cache directory entry, still a small percentage space overhead. The implementation also requires extra logic in the cache controller to handle message mode. However, this logic is relatively simple because message mode simply entails actions already performed by the cache controller as part of implementing the shared memory consistency protocol and does not introduce new types of actions.

2.2.4. *Advantages*

Message-oriented memory consistency reduces the number of bus transactions required to send a message compared to conventional memory consistency. The processor simply flushes an update down the memory system and allows the update to propagate through the memory system to other caches. In contrast, with conventional consistency, the sender of a message has to invalidate the corresponding cache line in each receiver processor's cache before writing the message and a receiver would, on reference to the new message, generate a cache miss that would have to acquire shared ownership of the cache line as well as the cache data from the sending cache. The bus transactions for a message transfer from sending to receiving processor for conventional coherency and for message-oriented consistency are illustrated in Figure 5. The conventional memory coherency protocol incurs extra bus transactions and extra roundtrip delays because of the request-response nature of this protocol whereas message-oriented consistency uses the "push" model of communication where data is sent without being requested. Moreover, message-oriented consistency minimizes the interference between source and destination processors. The sending processor is not delayed to gain exclusive ownership of the cache line and the receiving processors are not delayed by cache line flushes to provide consistent data. In contrast to other relaxed memory consistency models such as Stanford's DASH release consistency [Gharachorloo et al. 1989], message-oriented consistency reduces the base number of bus transactions and invalidations required, rather than just reordering them or allowing them to execute asynchronously.

The message-oriented memory consistency semantics also allow a simple network implementation of a shared channel segment between two or more network nodes. In particular, an update generated at one node can be simply transmitted as a datagram to the set of nodes that also bind the affected shared segment. As a separate project, we have built and used a 256 megabit per second fiber optic network that implements this transfer behavior between multiprocessor nodes. The best-efforts, but unreliable update semantics of message-oriented consistency obviates the complexity of handling retransmission, timeout, and error reporting at the memory level. A large-scale multiprocessor configuration can similarly afford to discard such bus and interconnection network traffic under load or error conditions without violating the consistency semantics. Section 3 describes techniques for building reliable communication on top of this best-efforts communication support.

With hardware support for direct cache-to-cache transfer, the source processor's cache could directly transfer the cache line toward the recipient processors' caches, thereby providing data transfer and notification in a single bus transaction

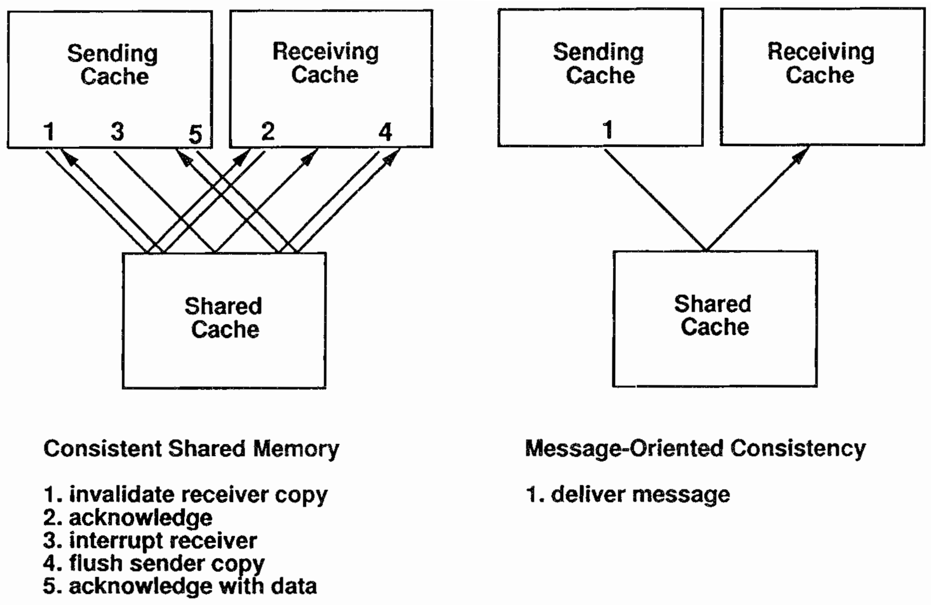

Figure 5. Reduced Memory Traffic for Message-Oriented Memory Consistency.

when the processors are on the same bus. This mechanism avoids the cache line transfer that normally follows the bus transaction to deliver the signal (causing invalidation and rereading of the cache line from the next level of memory or cache). This direct cache-to-cache delivery would reduce the latency and reduce the number of bus transactions to deliver the message. However, delivering the message directly into the first-level cache of a processor has several disadvantages. First, a message, especially a large one, can cause replacement interference with other cache lines in the cache, degrading the performance of the processor overall, especially if it does not handle the message immediately. This delivery method can also stall the processor by contending with the processor for access to the L1 cache during message delivery. Finally, direct transfer into the cache complicates the cache controller logic significantly.

The direct transfer into the L1 cache does not provide much benefit because the time required to trap in the message from the second-level cache is minimal. However, direct transfer between two networked nodes is used (with our network interconnection scheme) because of the high cost of a callback to get the data after receiving a signal. It is a cost-performance trade-off to decide between invalidation and direct "eager" transfer for the intermediate levels of cache interconnection.

As a further advantage of message-oriented consistency, its "push" model of update allows the signal notification to be piggybacked on the data transfer and update notification rather than requiring a separate signal bus transaction across the memory hierarchy, what we call *automatic signal-on-write*. This facility is described next.

2.3. Automatic Signal-on-Write

With automatic signal-on-write support, the signal is generated automatically by the action of the sending thread writing the message data. The sending thread then does not need to call the kernel `Signal` operation.

2.3.1. Implementation

Automatic signal-on-write is implemented as a relatively minor extension of the L2 and L3 cache controllers. If a writeback or flush operation is to a cache line in message mode, the cache controller generates a signal bus transaction on its interprocessor bus after allowing the write to complete. The signal bus transaction is straightforward to generate because the physical address is provided as part of the flush operation and the P_i bits in the associated cache tags indicate the L1 caches and processors to signal. If an L2 cache controller receives an update to a cache line in message mode or a notification of this update from the L3 layer, it similarly generates a signal bus transaction, propagating the signal to the processors using this cache.

In our implementation, there is both a signaling and a non-signaling message-oriented memory mode for cache lines, with only the former causing an interrupt. Using this mechanism, the signal-on-write can be programmed to occur only on the last cache line of a page (or any other multiple of the cache line), rather than strictly on each cache line.

The signal is always propagated down to the L3 level if necessary because the L3 entry corresponding to the flushed cache line is necessarily present because of the inclusion property: any line at L1 has the corresponding line loaded at L2 and L3. If a L2 cache controller signaled by the L3 controller does not hold the cache line corresponding to the physical address, the L2 cache controller signals all the processors. When a processor receives a signal on an address for which it has no signal threads, it clears its P_i bit for that cache line so it does not receive further signals on this line while this cache line is present at the L2 level. When none of the P_i bits for a cache line are set in a second level cache, the P_i bit for this L2 cache is cleared at the L3 level. If only one of the P_i bits are set on a cache line at the L3 level, the G bit is cleared for that cache line in that one L2 cache, eliminating the notification going to the L2 level. Migrating a signal thread from one processor to another means updating the cache tags to ensure delivery of the right signals to the new processor running this thread.

This design ensures that signals are always delivered to the per-processor signal FIFOs yet processors are not loaded with excessive numbers of signals that they do not want to receive (because they can shut off signals on a particular address after receiving a first extraneous one). However, processors do receive extraneous signal interrupts occasionally.

With this final optimization and transfer of cache data as well as notification down and back up the memory hierarchy, the update traffic matches in behavior and efficiency that of a specialized communication facility except it uses the memory busses rather than separate communication lines.

2.3.2. Advantages

Automatic signal-on-write reduces the overhead on a sending processor when no local thread is to receive the signal. In fact, in the case of the signal only being delivered to threads on other processors, the sending processor incurs no overhead for signaling the message beyond the operations required to flush the message.

Reduced sender overhead, in turn, shortens the latency of message delivery. In the absence of this facility, the sending thread must explicitly execute a kernel call, map the signal virtual address to a physical address, and then presumably access hardware facilities to generate the signal, or at least some interprocessor interrupt. In our experience, providing kernel-level access to the cache controller to explicitly generate a signal is more expensive in hardware than simply providing the logic in the cache controller itself to detect and act on flushes cache lines in message mode.

Automatic signal-on-write also allows the channel segment to specify the unit of signaling, transparently to the sending thread. For example, a receiver can pass the sender a channel that signals on every cache line or every page, depending on the receiver's choice. Without automatic signal-on-write, the sender needs to explicitly signal and thus needs to be coded explicitly for each signaling behavior, perhaps switching between different behaviors based on the channel type.

Optimized memory-based messaging was implemented in the ParaDiGM hardware and in an extended version of the V kernel. The memory-based messaging support replaced the previous kernel message support for RPC, resulting in significant performance improvement (see Section 4). This replacement also reduced the lines of code in our kernel by 15% and provided support for signals, which were not supported in our original system. The hardware support for memory-based messaging in ParaDiGM added approximately 6% to the logic of the cache controller. On a commercial machine, we estimate that the additional logic would easily fit within the cache controller ASIC and therefore would not increase manufacturing costs. Finally, the 128-entry FIFO buffer per processor added 6% to the per-processor cost, totaling from 12% to 21% of the CPU board costs [Hennessy & Patterson 1990]. The extra logic constituted less than 1% of the hardware cost for a complete four-processor board. Moreover, this hardware is needed in any case for large-scale interrupt support. Overall, we conclude that this approach reduces hardware costs compared to the multiple *ad hoc* interprocessor schemes used in many current systems.

The next section describes the user-level library implementing RPC in replacement for the previous kernel support.

3. Remote Procedure Call Implementation

The remote procedure call (RPC) facility provides an easy-to-use application interface to the raw memory-based messaging facility, which is fundamentally a difficult interface for applications to use directly for several reasons. First, signals can be lost so the client of the raw interface has to implement a reliable error control/recovery mechanism or else not require reliable signal delivery. Second, partially written messages can be received because an involuntary cache write-back can cause the L2 cache to perceive that the processor is sending the message. This situation can also arise with multi-cache line messages. Finally, as a consequence of signal and data loss, duplicate messages can be sent as well. To deal with these difficulties, the RPC system incorporates standard transport protocol techniques similar to, for example, TCP and an ONC RPC implementation on top of UDP.

The remote procedure call facility including the transport facilities is implemented on top of memory-based messaging in an application-linked run-time library. This approach contrasts with a kernel implementation as in V [Cheriton 1988.8] and Amoeba [Tanenbaum et al. 1990], and the separate network server implementation as in Mach [Accetta et al. 1986].

In overview of the RPC structure, there is a unidirectional memory-based messaging channel segment from the client to the server and a similar unidirectional return channel from server to client. To call the server, the client writes a message containing the RPC parameters and stub identification to the server's incoming channel segment and sends an address-valued signal to the server. Upon receiving the signal, the server processes the message by unmarshaling the arguments onto a stack, calling the appropriate procedure, as in a conventional implementation, writing the return values as a message on the return channel and signaling the client. The client then receives the signal indicating the response has been sent and unmarshals the response.

Currently in our prototype, the client establishes these channels with the server by contacting the server using a well-known shared channel segment associated with the server, similar to the well-known multicast addresses used in previous systems. On this first contact, the client provides request and response channel segments to which the server then binds, specifying a memory region and an RPC signal handler.

To ensure reliable delivery of RPC call and return messages, a checksum is used on the message to ensure that all portions of the message were received. Using "integrated layer processing" [Clark & Tennenhouse 1990], the checksum calculation cost is not significant when integrated with the data copy. The data copy is required as part of marshaling and unmarshaling parameters at either end of the channel segment. For instance, a checksum calculation with the copy operation on a 25 MHz DecStation 5000/200 adds 8% to the copy time. A message that fails the checksum is normally discarded, as in conventional transport protocols. The unmarshaling copy operation also prevents the received parameters from being overwritten while the receiver is processing the call. That is, because the call parameters have been moved from the message segment area, a subsequent overwriting message cannot affect the call processing. In fact, the checksum is calculated as part of the unmarshaling copy so that it can safely detect an overwrite of the message during the unmarshaling.

Lost or dropped messages are detected and handled by an asynchronous timer thread. The timer thread simply reviews the set of outstanding calls periodically and requests retransmissions if there has not been a response or acknowledgment within the timeout period. Because the timer thread operates asynchronously and independently of the message sending, the normal case of sending a call and getting a response when the packets are not lost operates without the timer overhead required in conventional transport protocols.

The messages are sequenced by writing each successive message to the next location in the message channel following the previous one, wrapping to the beginning of the channel region after hitting the end of the channel segment. Each channel wrap increments a wrap count that is used as the base for the checksum calculation so if a receiver is out of synchronization with the sender, it sees the messages as corrupted because the checksum fails.

Using a full transport protocol for local (or inter-address space) calls is a novel and contentious aspect of our approach. There is the obvious concern that this approach unnecessarily degrades local communication performance. However, it is the preferred approach with memory-based messaging for several reasons. First, the cost of the transport-level mechanism is not significant in the local case. The cost of a simple checksum calculation such as used with TCP is dominated by the memory access time to the data. By integrating this calculation with the marshaling and demarshaling, as described earlier, the cost is hidden by the memory access latencies of the copy operation, at least on modern RISC processors. Moreover, for small messages, the checksum cost is insignificant compared to the other remote procedure call run-time overheads, such as scheduling the executing thread for the procedure. The frequency of large messages for which the copy and checksum time is significant is reduced by memory-mapping most of the I/O activity. In fact, the large messages that are not subsumed by the memory-mapped I/O approach are predominantly communication with a network file server, for which the checksum overhead is required, as in conventional systems. Moreover, bulk flows like video do not require the same degree of reliable delivery and thus can be transmitted without checksums.

The other significant transport mechanism, timeout for retransmission, executes asynchronously to the call and return processing so, in the normal no-loss case, the overhead is a small fixed percentage of the processor time. This cost can be made arbitrarily low by increasing the timeout parameter. That is, with a timeout of T seconds and processing cost P seconds, the processing overhead is PT per second. With signal loss extremely rare in the expected case, the cost in system performance is dominated by the time to timeout and retransmit.[6]

Second, implementing reliable transport for local calls allows the RPC run-time library to use the same mechanism for local and remote calls, avoiding the overhead and complexity of checking whether a channel is local or remote on each call. Channel segments appear the same to the software outside the kernel whether they connect within a machine or span multiple machines. Moreover, because a channel can be rebound during its lifetime so that its endpoint is remote rather than local, the RPC mechanism would need to either check with the kernel on each call or be reliably notified of the rebinding if it did not incorporate reliable transport always as we have done. The former would incur a significant overhead, estimated to be comparable to the 8 percent overhead we measure for the checksum calculation on the common "small" RPCs, obviating the benefit of discriminating between local and remote segments. The latter approach requires additional code complexity in the RPC mechanism.

Finally, providing software support for reliable delivery allows the hardware in large-scale multiprocessors to be much simpler. For example, in our hardware

6. To deal with the potential of dropped signals from devices, our device drivers periodically check the device interface for activity rather than requiring the device hardware to retransmit.

implementation, signals can be lost because of a local FIFO buffer overflow, although this is unlikely. Preventing overflows in hardware would require some form of flow control. Flow control is difficult to do across a large-scale interconnection network, and is virtually impossible with multicast communication. A significant source of cost and complexity in (for example) the CM-5 communication network is the hardware to ensure reliable message delivery. Moreover, even in such hardware schemes, there is still a software overhead to check for overflow conditions. Therefore, providing a full transport mechanism in the local case reduces the requirements and cost of the hardware, and allows an application to tolerate more hardware faults.

The remote procedure and transport mechanism is implemented in our system as a C++ class library executing in the application address space, and is well-structured for specializing for particular applications, including those that do not require full reliability. A basic channel mechanism in the class library supports a raw form of communication and does not impose the transport level overhead for this type of traffic. For instance, a channel segment is well-suited for real-time multicast datagram traffic like raw video because data units are being rapidly updated by the source, and the occasional dropped cache line unit or lost signal does not significantly affect the quality of the resulting picture. Note that dropped cache line updates do not put the data out of sequence in any sense because each cache line is specifically addressed with its local address within the channel segment. Derived classes of the basic channel class provide the reliable transport mechanisms described above.

There are three issues we continue to work in this RPC implementation. First, large messages generate a signal interrupt on each cache line unit of data, causing extra overhead compared to a single signal interrupt at the end of a message. A means of indicating the last cache line of message data as part of the flush operation appears to eliminate this overhead. The channel knows the beginning of the message from the state maintained by the channel.

Second, there is a need to select the channel region size as part of setting up the memory-based messaging channels for RPC. Logically, the channel region size is effectively the flow control window size. However, our current implementation does not allow the transmission of a call or return message that is larger than the region size. This limitation is analogous to the size limit imposed by simple RPC protocols used over UDP, where the limit is the 64 kilobyte limit of UDP/IP packets. Allowing larger call and return messages would introduce the potential of the caller or the callee being blocked indefinitely by flow control in the middle of transmission, just as can occur with large RPCs over TCP. Finally, memory-based messaging is essentially connection-oriented in nature, incurring the overhead of setting up and tearing down connections, contrasting with the datagram and dynamic binding of many other IPC systems. We have been experimenting with techniques for sharing channels for low-performance communication and only creating separate channels for performance-critical cases. Happily, the primary resource used by a channel is virtual memory, which can be paged out when not being used just like other portions of virtual memory. With a virtual memory system capable of handling very large programs, the limit on number of channels is purely the limit on the amount of virtual memory, which should be very large, so channel teardown is almost never required.

In summary, the best-efforts reliability of our memory-based messaging support allows better performance with scale at a lower hardware cost, transfers the complexity of ensuring reliable communication (when needed) from hardware to software and avoids having separate mechanisms in the application space for local and remote communication. The next section provides some performance measurements of our implementation.

4. Performance Evaluation

The performance of optimized memory-based messaging was evaluated using an extended version of the V distributed system [Cheriton 1988.7] and the ParaDiGM multiprocessor [Cheriton et al. 1991]. The specific configuration is an 8-processor shared memory multiprocessor configuration consisting of two multiprocessor modules each containing four Motorola (25 Mhz) 68040 processors sharing an L2 cache that supports our optimizations. As in Figure 3, multiple multiprocessor boards share an L3 cache, where the consistency is controlled by kernel software. Although this hardware that we designed and implemented is not the fastest available at this time, we argue that the logical design is applicable to much faster processors, and a faster processor would not significantly reduce the benefits of our optimizations (see Section 5).

4.1. Hardware Performance Benefits

To evaluate the benefits of the hardware optimizations, we developed a software-only implementation of memory-based messaging as a basis for comparison. In this implementation, the sender traps to the kernel, and uses a queue and interprocessor interrupt to notify the receiver of the signal.

Table 1 compares this software-only version with our optimized messaging implementation, listing the execution times (and MC68040 instruction counts) of various kernel and user-level components for these two implementations. These

Component	Software-Only		Hardware-Supported	
	Time	Instr.	Time	Instr.
sender system call	3	13	–	–
virtual-to-physical mapping	3	16	–	–
determine receiving processors	4	23	–	–
insert in kernel queue	6	55	–	–
generate interrupt	1	4	–	–
get physical address from FIFO	–	–	2	11
remove from kernel queue	6	45	–	–
physical-to-virtual mapping	1	9	1	9
invalidate L1 cache lines	–	7	1	7
check if kernel is receiver	1	4	1	4
signal function scheduling	6	37	6	37
return to user code	4	25	4	25
user-level state save/restore	1	11	1	11
Total	36	249	16	104

Table 1. Hardware-Supported vs. Software-Only Implementations (Time in μsecs)

measurements show that using all three hardware optimizations provide a factor of two reduction in kernel overhead even in a small-scale system. This reduction is achieved by hardware support that eliminates the instructions required to deliver the signal value to the appropriate processor. The cost of message delivery from the signaled processor to specific threads would be reduced significantly in the common case using a reverse TLB and user-mode signal trapping. (See Section 2.1.3)

Section 5 shows that even greater benefit can be expected for future larger-scale systems, because message delivery using the shared data structures of the software-only implementation becomes more expensive with a larger-scale shared memory system.

4.2. Remote Procedure Call Measurements

Table 2 provides a breakdown of the components of the RPC implementation using optimized memory-based messaging, not including the memory-based messaging costs detailed in Table 1. The majority of the time is spent on marshaling and demarshaling. The mapping between object and channel, and vice versa, is the other major component.

Component	Request/Reply Time
map from object to channel	1
marshal 32-bytes	2
trigger signal	1
map from channel to object	1
unmarshal 32-bytes	2
Total	7

Table 2. RPC Component Timings (μsecs)

System Call	Execution Time
Create Segment	582
Bind Memory Region	320
Enable Signal	249
Disable Signal	231
Unbind Memory Region	243
Release Segment	636

Table 3. Setup Cost Timings (μsecs)

The total latency of a 32-byte RPC between two processors sharing an L2 cache is 47 μsecs. This performance is 2.6 times faster than the software-only version of memory-based messaging RPC, which takes 124 μsecs.

A 32-byte RPC between processors, on separate L2 caches, sharing an L3 cache takes 127 μsecs ("Opt. MBM (L3)" in Table 4). The corresponding software-only RPC takes 1860 μsecs. (Both L3 times are somewhat inflated because of the partially-optimized software L3 cache consistency support in the current implementation.)

These measurements do not include the costs of creating and destroying the channel segments and binding them into the memory of the respective address spaces, as required before RPCs can be executed. In our object-oriented RPC implementation, the setup is performed as part of creating a local proxy object. Table 3 provides the basic setup and tear-down costs. Summing the execution time column (omitting disable signal because it is subsumed by unbinding the memory region), connecting to a new object and then disconnecting can take 2030 μsecs. Thus, a significant number of RPCs need to be performed over a channel to amortize this overhead to a small percentage. In earlier measurements of V [Cheriton & Williamson 1987], we observed a high degree of persistence in communication between clients and servers, and a small number of such pairings. Thus, we expect this setup overhead to be acceptable, if not insignificant, when amortized over the

System	Null RPC	Send, Recv 32 bytes	Send 1KB	Processor	MIPS
Opt. MBM (L2)	44	47	215	68040	15
Opt. MBM (L3)	120	127	502 (est.)	68040	15
Soft MBM (L2)	121	124	268	68040	15
Soft MBM (L3)	1857	1860	12580	68040	15
Mach 3.0	95	98 (est.)	268 (est.)	DEC 3100	14.3
V System	469 (est.)	472	794	68040	15
URPC	93	899 (est.)	608 (est.)	Firefly	3
LRPC	125	131 (est.)	640	Firefly	3

Table 4. Comparative RPC Timings (μsecs)

typical number of RPCs that use a channel segment during its lifetime. However, the setup time should definitely be factored into the RPC time for applications with many short-lived connections.

4.3. Comparison with Previous Systems

For comparison, Table 4 shows published RPC times for previous message-based operating systems. These measurements indicate that optimized memory-based messaging RPC (labeled "Opt. MBM (L2)") is clearly faster than the original V system. The V performance suffers from several factors. First, the V copy model of messaging imposes a copying overhead that is not present with memory-based messaging. Second, there are many "on-the-fly" actions performed on each RPC because there is no connection setup prior to sending a V message. These actions are eliminated by the connection setup with memory-based messaging. Finally, the V messaging requires a context switch during the RPC.

The optimized memory-based messaging is also faster than Mach 3.0 (based on our estimates from published figures for the 32-byte and 1-kilobyte messages). Mach 3.0 has a connection-oriented model based on ports but still suffers from copy cost and context switching overhead.

The URPC and LRPC systems appear to be the most competitive with optimized memory-based messaging mechanism. In fact, if one purely scales based on rough MIPS ratings, one might conclude that URPC system is faster. However, we believe there are several considerations that still favor optimized memory-based messaging. First, the published URPC time can only be achieved when the server constantly polls client message channels and manages to find a client message immediately after the client queues it. This polling mechanism does not appear practical in real systems where a server would have a large number of clients. Moreover, the time to locate the particular requesting client would be larger even if polling was used with a busy server. Second, both LRPC and URPC reduce the number of copy operations by using parameters directly from the shared segment, eliminating the unmarshal step. However, this technique relies on using the VAX's separate argument stack, a mechanism not supported by modern RISC processors. Finally, the overhead of URPC shared memory references to control the server's queue would make URPC substantially slower than our optimized messaging for calls between most pairs of processors in a large-scale multiprocessor system because of the memory coherency cost. The LRPC and URPC measurements were done on the VAX-based Firefly multiprocessor on which a reference to a write-shared datum incurs essentially the same cost as a private memory reference because of the write-broadcast update protocol and the slow processors relative to the memory system. However, a similar reference on a machine like ParaDiGM, DASH [Gharachorloo et al. 1989], KSR-1 and numerous forthcoming architectures from Cray, Convex, and others would cost approximately 100 cycles or more, assuming the referenced data was last updated by another processor. Besides increasing the latency, these shared memory references also impose an extra load (not present in optimized memory-based messaging) on critical resources such as memory busses.

These measurements show that optimized memory-based messaging is competitive with the fast RPC implementations of other systems. Moreover, memory-based messaging also provides data streaming between address spaces at memory system performance, a facility not directly supported by the other communication approaches. The next section shows that these benefits are even more significant for future (large-scale) system configurations.

5. *Benefits in Future Systems*

The performance benefits for optimized memory-based messaging were estimated for future larger and faster machines using a simple simulation. This simulation incorporates a cost model, based on the factors we have identified in our implementation, with the actual costs scaled for the expected hardware parameters. Using this simulation, a software-only implementation of memory-based messaging was compared with configurations introducing each hardware enhancement. The case measured is a message of 32 bytes, a cache line, sent to another processor, where a null signal function is executed. In the simulations without hardware support, address-valued signaling was performed using a software-controlled global queue to hold the virtual address of the message. Similarly, conventional shared-memory

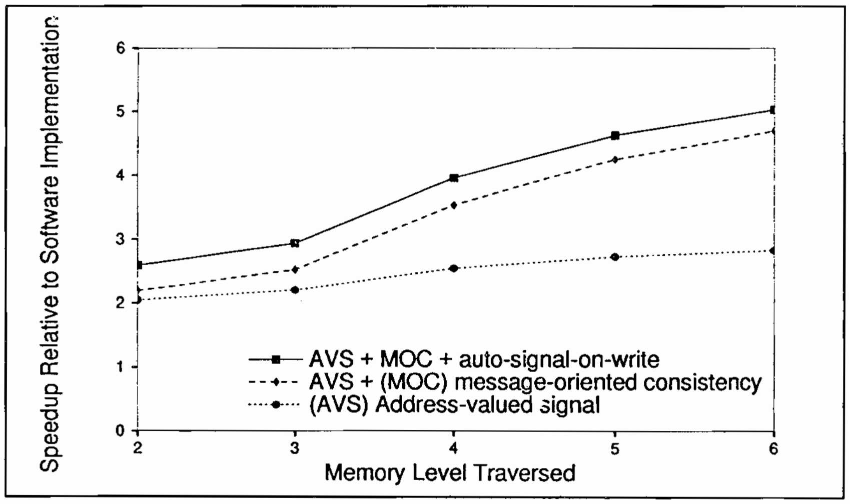

Figure 6. Speedup of 32-byte cache line transfer vs. memory levels traversed (25 MHz Processor).

consistency using a write-invalidate was assumed in place of message-oriented consistency. Without hardware support for automatic signaling on write, the model assumes that, after the message write, the sender traps to the kernel to execute the `Signal` system call. (It could also generate the call after trapping on a reference to a write-protected memory location, a technique used to emulate automatic signal on write.)

For these simulations, we measured our L1 fill time to be 1.12 μsecs for a 32-byte cache line. An estimated hardware-supported L2 fill time is 3.36 μsecs. A single hop across our fiber optic link transfer 32 bytes is 6.1 μsecs.

Figure 6 shows the speedup of a message transfer for optimized memory-based messaging compared to a software-only implementation as a function of the distance traveled by the message. The values of 2 and 3 on the x-axis correspond to a message delivered through an L2 cache and L3 cache respectively. The values of 4, 5, and 6 correspond to one, two and three hops across a fiber optic link.

The speedup is more significant for processors widely separated in the memory system because the transfer is dominated by the cost of the bus/network transactions. Address-valued signaling and message-oriented consistency reduce the number of such transactions compared to conventional shared memory techniques, as was illustrated in Figure 5. Note that the number of transactions in a conventional system on the L3 bus and network is effectively twice that of the L2 level because of the use of split-transaction protocols in the lower cache levels. Thus, the savings from message-oriented consistency are greater for these levels than the L2 level, both in reduced transactions as well as reduced latency.

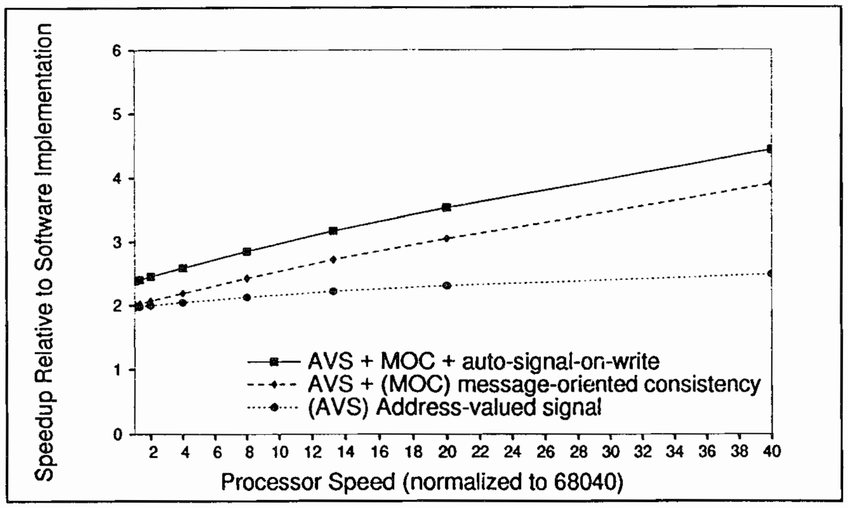

Figure 7. Speedup of 32-byte cache line transfer vs. processor speed (through shared L2 cache).

The message transfer speedup between processors local to an L2 cache results from the reduced software overheads of address-valued signaling and automatic-signal-on-write. The benefit of the message-oriented consistency is reduced by the relatively high speed of the L2 bus in this case.

Approximately 79% of the speedup of a message transfer through an L2 cache and 56% of the speedup through 3 hops of the fiber optic link is attributable to hardware support for address-valued signaling. Message-oriented consistency accounts for 6% of the speedup of an L2 transfer and 37% through 3 fiber optic links. Automatic signal-on-write support accounts for 15% of the speedup of an L2 transfer and 7% through the 3 fiber optic links.

Figure 7 shows the benefits of the optimizations as a function of the processor speed. The processor speed is normalized to the speed of the 68040. A faster processor is assumed to use a correspondingly faster L2 bus. The base or 68040 L2 bus transfer time in this simulation is one 32-byte cache line in 0.3 μsecs. A 32-byte message is assumed to transfer over a fiber optic link in 2 μsecs.

The increase in speedup with increasing processor speed in Figure 7 shows that faster processors simply emphasize the memory system latencies, even with a high-speed L2 bus. At higher processor speeds, the costs of the software operations, such as physical to virtual address mapping, diminish, affecting both implementations equally but leaving the relative speedup unchanged.

Figure 8 shows the speedup for a 32-byte message transfer as a function of processor speed over a fiber optic link. This figure shows that the memory system overhead is again more apparent with faster processors. While faster processors

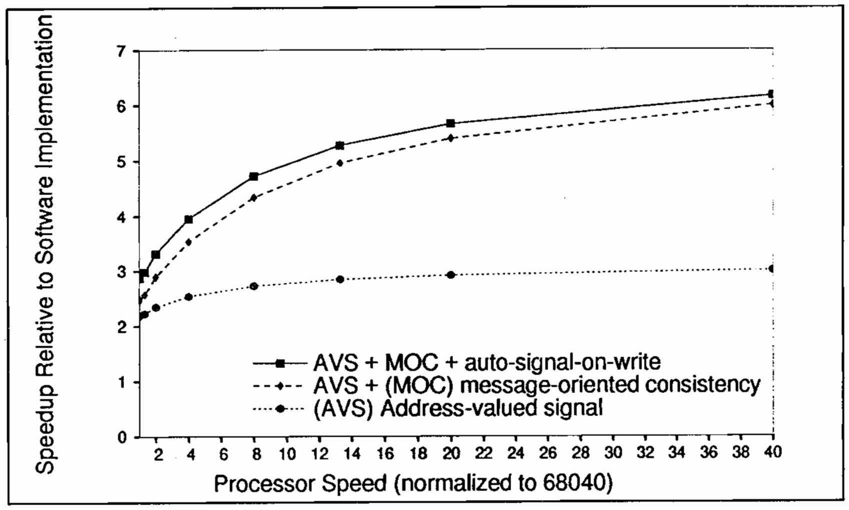

Figure 8. Speedup of 32-byte cache line transfer vs. processor speed (fiber optic link).

allow faster message transfers, these gains are limited by the latencies of a large-scale memory system. Optimized memory-based messaging minimizes the actions required of the memory system, thereby providing scaling with increasing processor speed.

Overall, our simulation of larger and faster architectures suggests that optimized memory-based messaging provides even greater benefits on these future machines because it optimizes for the bottleneck resources, namely the memory system and its supporting interconnect. Moreover, optimized memory-based messaging obviates the need for conventional interprocessor interrupts, separate message mechanisms and even I/O subsystem hardware if the I/O devices were designed to interface to the memory-based messaging system.

6. *Related Work*

The original architectural support for optimized memory-based messaging was described by Cheriton et al. [Cheriton et al. 1989] in a design that was refined and implemented as the ParaDiGM architecture [Cheriton et al. 1991]. While the basic design has remained largely the same, a number of refinements were made as part of the ParaDiGM implementation and measurements. As an example, we discovered that it was faster to invalidate a received cache line in software than to have the cache controller perform this task.

The basic memory-based message model is similar to that used in the Berkeley DASH project [Tzou & Anderson 1991], URPC [Bershad 1990], and many commercial systems using shared memory for communication between processes. Our contribution has been the refinement of the signaling and consistency support and an efficient hardware and software implementation that further optimizes this communication model.

The signaling mechanism has some similarity to the signal mechanism in Unix. However, the extension to *address-valued signaling* provides a translated address optimized for messaging while providing a mechanism sufficient to implement Unix signals. That is, a well-known range of memory addresses could be allocated for Unix signals, with an address for each Unix signal number.

An alternative approach to our memory-based messaging approach is to provide a separate hardware communication facility. For example, the Alewife multiprocessor design [Chaiken et al. 1991; Kranz et al. 1992] provides messaging support directly into the processor. The network interface supporting messaging is connected to the processor using the coprocessor interface on the SPARC-1 processor. The network interface supports a DMA engine, a sliding message buffer window and specialized coprocessor instructions. Because this design allows messages to transfer directly from and to the processor using the co-processor interface and thus bypass the memory system, it should in principle be faster than our approach (no measurements of Alewife were available at the time of writing). However, the Alewife approach depends on the existence of a co-processor or similar interface devoted to messaging. Very few processors have such an interface. Moreover, with the limits on chip pin count being an issue, it is more performance-effective to use these pins for wider access to memory than dedicating them to only communication support.

More generally, we see our approach of integrating messaging support into the memory system as directly benefiting from each improvement in memory system performance, rather than contending with the memory system for pin count and design cycles. This is in line with the key focus of computer architects. That is, the cost of moving data between a source and a recipient is primarily limited by the cost of writing the data to the shared segment and reading the data back in the receiver process. The mainstream computer architecture community emphasizes optimizing the memory system rather than the communication facilities for several (good) reasons. First, memory is well recognized as the primary bottleneck to the performance of fast RISC processors. Most applications make far greater demands on the memory system than on the communication facilities so it is better to use the chip real-estate for larger on-chip caches that special-purpose communication support. Second, most systems are small in scale so there is limited market for machines that really require specialized communication support. In fact, for the foreseeable future, the number of multiprocessors will almost certainly remain vanishingly small compared to the number of uniprocessor systems. Finally, most processor designers avoid less-proven and less standardized features, if for no other reason than to minimize design time. With the cost, reliability and time-to-market issues strongly driving this market, hardware and operating systems designers are expected to remain focused on basic memory system performance and are unlikely to have resources left for effort that purely improves communication performance.

The other general alternative approach is a pure software implementation of interprocess communication. Most previous performance work here has focused on reducing the cost as close as possible to the raw copy cost (e.g., V [Cheriton 1988.8], Amoeba [Tanenbaum et al. 1990] and Taos [Schroeder & Burrows 1989]) and to reduce the copy cost itself (e.g., Mach [Accetta 1986] and URPC [Bershad 1990]). Mach uses the copy model of IPC and optimizes it using memory mapping techniques, whereas the memory-based messaging approach takes the memory mapping model and extends it for efficient communication. We believe that the cache and interconnection structure of modern computer memory systems makes the copy model of messaging inadequate, especially for high-performance communication applications such as multi-media, simulation and high-performance I/O. Optimized memory-based messaging, as one alternative, provides better and more scalable performance even without hardware support.

Memnet [Delp 1988] is another system that provides a memory model of communication. However, it also uses special and separate communication hardware, using a consistency mechanism to drive network transmissions to provide the illusion of a memory module shared by all the machines on the network. This approach duplicates the memory system, at least for a shared memory multiprocessor, in a specialized communication subsystem and then makes it look like memory. Thus, Memnet is the opposite to our approach, both in terms of model and mechanism, of integrating the communication into the memory system.

A number of other systems provide a memory interface to communication facilities. However, these systems are of a significantly different genre. For example, the Xerox Alto, the original SUN workstation, the CM-5, and many other systems provide a location in memory to read and write to receive and transmit network data. However, this approach is generally just providing a memory port to a separate conventional communication mechanism which is not really integrated with the memory system. In particular, each write operation to the communication interface requires an uncached write operation over the interconnecting bus, rather than using the cache line block transfer unit, as we have used.

7. Conclusions

Optimized memory-based messaging is an interprocess communication facility that is simple to use (through the RPC interface), inexpensive to implement in software and hardware, and significantly faster than the interprocess communication support provided by conventional operating systems and hardware. It appears that its advantages may be even more significant in large-scale multiprocessor systems expected in the future.

The approach of providing communication in terms of the memory system has simplified both the hardware and the software. The software implementation largely consists of extensions to the basic virtual memory mechanisms already provided by the operating system kernel. For example, the signaling mechanism uses the same data structures to map to recipients of a signal as the virtual memory system uses for mapping addresses and the same signal delivery used for virtual access signals (similar to `SIGSEG`) in Unix. With our operating system kernel, this approach is the only communication and I/O facility provided, thus eliminating the buffering, queuing, synchronization and mapping code and data structures used in most message-based operating system micro-kernels. These benefits were realized even more strongly in the V++ Cache Kernel [Cheriton & Duda 1994] which we developed subsequent to the work reported here.

The hardware support is a simple, low-cost extension to the directory-based memory caches that are increasingly common with shared memory multiprocessor machines. The three refinements of address-valued signaling, message-oriented consistency and automatic signal-on-write complement each other to further simplify the hardware and improve performance. Based on our implementation, we estimate the additional hardware support costs to be less than 1% of a multiprocessor board as used in ParaDiGM and even less of the overall system cost. Thus, the hardware support is affordable even for small-scale multiprocessors where the performance benefits are the least. Moreover, the address-valued signaling provides a unified model for an interrupt system supplanting the specialized facilities for device interrupting, interprocessor interrupts, and hardware communication facilities that are present on some systems.

Our measurements of our software/hardware system show performance that compares favorably with other high-performance interprocess communication facilities. Using simple performance models, we have estimated that hardware-supported memory-based messages would offer approximately a three-to-five fold improvement in performance for basic communication operations on moderate to large-scale multiprocessor systems.

The *memory-based messaging* model exports an interface that allows an efficient remote procedure call implemented outside the kernel that supports both local and over-the-network communication. It can also be used with specialized communication software to support high-performance real-time communication for video and graphics where the reliability and structure of RPC are not needed. The address-valued signaling mechanism can also be used as an object notification mechanism between threads and address spaces independent of the messaging use.

As part of on-going work, we are addressing several issues. First, we are experimenting with different schemes for efficiently mapping the RPC mechanism onto memory-based messaging to allow specialization of RPCs for particular situations [Zelesko & Cheriton 1996]. For example, we are experimenting with a non-blocking RPC with no return value, optimized for some distributed simulations. More generally, we would like to use the memory-based messaging facility for less structured communication as well, such as video and audio channels. Second, we are also investigating the issues of moving large amounts of data using optimized memory-based messaging. In our prototype, the receiving processors take an interrupt on every cache line. A means is required for the sending thread to indicate the last cache line that is part of a message and have the signal mechanism only deliver an interrupt on this last cache line. Finally, we are continuing to develop network hardware and channel management software to extend the memory-based messaging over network links with non-trivial topologies. This hardware would also benefit from a "last cache line of message" indication, so it could transmit a sequence of cache line flushes as single packet, reducing the per-packet overhead compared to sending each cache line as a separate packet, as we currently do. Sending larger packets would also make this networking technology compatible with different cache line sizes, allowing its use across more hardware platforms.

Overall, based on our experience to date, optimized memory-based messaging appears to be a promising approach for achieving cost-effective high-performance communication in future systems. The central theme of our work is the integration of communication support with the memory system model and mechanism. This approach reduces the specialized system primitives and complexity of conventional approaches to communication and provides performance gains in communication by capitalizing on the well-motivated drive to improve memory system performance. From our experience to date, we judge this approach as superior to approaches that provide communication as a separate mechanism.

8. Acknowledgments

This work was sponsored in large part by the Defense Advanced Research Projects Agency under Contract N00014–88–K-0619. Robert Kutter was supported by IBM. Kieran Harty developed the virtual memory implementation and provided valuable assistance in the implementation of address-valued signaling. Christophe Metivier developed the prototype hardware. Mendel Rosenblum, John Chapin, Hugh Holbrook, and Sandeep Singhal provided valuable comments during the development of this paper. Wayne Mesard developed the kernel software that supports the cache protocols at the L3 level.

References

1. Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, Mach: A New Kernal Foundation for UNIX development, *USENIX Association Summer Conference Proceedings*, pages 93–112, USENIX, June 1986.
2. Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Inc., 1986.
3. Brian N. Bershad, *High Performance Cross-Address Space Communication,* Ph.D. thesis, University of Washington, Department of Computer Science and Engineering, June 1990.
4. David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron, Translation Lookaside Biffer Consistency: A Software Approach, *3nd Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–121, ACM, April 1989.
5. David Chaiken, John Kubiatowicz, and Anant Agarwal, LimitLESS Directories: A Scalable Cache Coherence Scheme, *4th Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234. ACM, April 1991.
6. David R. Cheriton and Cary Williamson, Network Measurement of the VMTP Request-Response Protocol in the V Distributed System, *SIGMETRICS ACM*, 1987.
7. D. R. Cheriton, The V distributed operating system, *Communications of the ACM*, 31(2):105–115, February 1988.
8. David R. Cheriton, The V Distributed System, *Communications of the ACM*, pages 314–333, March 1988.
9. D. R. Cheriton, H. A. Goosen, and P. D. Boyle, Multi-level shared caching techniques for scalability in VMP-MC, *Proc. 16th Int. Symp. on Computer Architecture*, pages 16–24, May 1989.
10. David R. Cheriton, Gregory R. Whitehead, and Edward W. Sznyter, Binary Emulation of Unix using the V Kernal, *Usenix Summer Conference*, Usenix, June 1990.
11. D. R. Cheriton, H. A. Goosen, and P. D. Boyle, ParaDIGM: A Highly Scalable Shared Memory Multicomputer Architecure, *IEEE Computer*, 24(2):33–46, February 1991.
12. D. R. Cheriton and K. J. Duda, A caching model of operating system kernal functionality, *Proceedings of 1st Usenix Symposium On Operating System Design and Implementation (OSDI)*, pages 215–228, ACM, November 1994.
13. D. D. Clark and D. L. Tennenhouse, Architectural Considerations for a New Generation of Protocols, *Computer Communication Review*, pages 200–228, ACM, September 1990.
14. G. S. Delp, *The Architecture and Implementation of Memnet: A High-Speed Shared-Memory Computer Communication Network*, Ph.D. thesis, University of Delaware, Department of Electrical Engineering, 1988.

15. David Kranz et al., Integrating Message-Passing and Shared-Memory: Early Experience. *SIGPLAN Notices*, 28(1), September 1992.

16. Robert P. Fitzgerald, *A Performance Evaluation of the Integration of Virtual Memory Management and Inter-Process Communication in Accent*, Ph.D. thesis, Carnegie-Mellon University, Department of Computer Science, October 1986.

17. Kourosh Gharachorloo, Anoop Gupta, and John Hennessy, Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors, *3rd Int. Conference on Architechural Support for Programming Languages and Operating Systems*, pages 113–121, ACM, April 1989.

18. Per Brink Hansen, *Operating System Principles*, Englewood Cliffs, N.J., Prentice-Hall, 1973.

19. Joe Heinrich, *MIPS R4000 Microprocessor User's Manual*, Prentice-Hall, Inc., 1993.

20. John L. Hennessy and David A. Patterson, *Computer Architecure: A Quantitative Approach*, Morgan Kaufman Publishers, Inc., 1990.

21. C. May, E. Silha, R. Simpson, and H. Warren, *The PowerPC Architecture*, Morgan Kaufmann, Inc., 1994.

22. David A. Patterson, Garth Gibson, and Randy H. Katz, A Case For Redundant Arrays of Inexpensive Disks (RAID), *SIGMOD*, pages 109–116, ACM, June 1988.

23. Bryan S. Rosenberg, Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors, *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 137–146, ACM, December 1989.

24. Michael D. Schroeder and Michael Burrows, Performance of Firefly RPC, *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 83–90, ACM, December 1989.

25. A. S. Tanenbaum et al., Experiences With the Amoeba Distributed Operating System, *Communications of the ACM*, 33(12):46–63, 1990.

26. J. Torrellas, M. S. Lam, and J. L. Hennessy, Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors, *Workshop on Scalable Shared-Memory Architecture*, Seattle, May 1990.

27. Shin-Yuan Tzou and David P. Anderson, The Performance of Message-Passing Using Restricted Virtual Memory Remapping, *Software-Practice and Experience*, 21(3):251–267, 1991.

28. M. J. Zelesko and D. R. Cheriton, Specializing Object-Oriented RPC for Functionality and Performance, *Proceedings of the 16th International Conference on Distributed Computing Systems*, IEEE, May 1996.