

An Efficient, Protected Message Interface



The authors describe a message interface that provides high performance and low processor overhead, and features a robust protection model. They discuss this system in the framework of the multithreaded MIT M-Machine and show that—unlike other approaches—this system is able to avoid starvation while providing protection and maintaining high efficiency.

Whay Sing Lee

William J. Dally

Stephen W. Keckler

Nicholas P. Carter

Andrew Chang
Stanford University

In traditional message interfaces, high latency and processor occupancy inhibit our ability to exploit large-scale parallelism. Even though recent designs address this problem by removing OS layers from the interface,¹⁻⁵ the remaining overhead is still large. To amortize communication overhead of hundreds of cycles, programmers use messages that are hundreds to thousands of words in size. Consequently, threads run for thousands of cycles between communications, which precludes many parallelization opportunities.

When designers incorporate multiple hardware thread slots onto each node, this overhead is exacerbated if primitive support for fair and protected resource allocation is lacking. Much of the communication overhead can be removed by carefully making complementary design choices in primitive messaging mechanisms in order to facilitate messages as short as several words in size and to enable fine-grain parallelism. For instance, a complete round-trip null message takes only 38 cycles in the MIT M-Machine, an experimental multicomputer designed to exploit parallelism with a wide range of granularity.⁶

The design space for messaging mechanisms can be divided into three elements:

- *mapping*, which defines how the network interface (NI) hardware is presented to the software;
- *atomicity*, which determines whether message injection/extraction is uninterruptible; and
- *dispatch*, which describes the mechanism that determines how the processor reacts to message arrivals.

Memory-mapped interfaces access the network state through specific addresses as if they are part of the memory, while instruction/register-mapped interfaces integrate tightly with the processor, injecting and extracting messages with special instructions or reading and writing special device registers. A streaming interface allows the message's head to worm through

the system without waiting for its tail. Conversely, a buffered interface requires each message to be completely stored in a buffer before it is injected in an uninterruptible fashion or received as an atomic unit. Conventionally, an interrupt mechanism asynchronously displaces the current program with an interrupt handler when the message arrives, while a polling system periodically checks for message arrival.

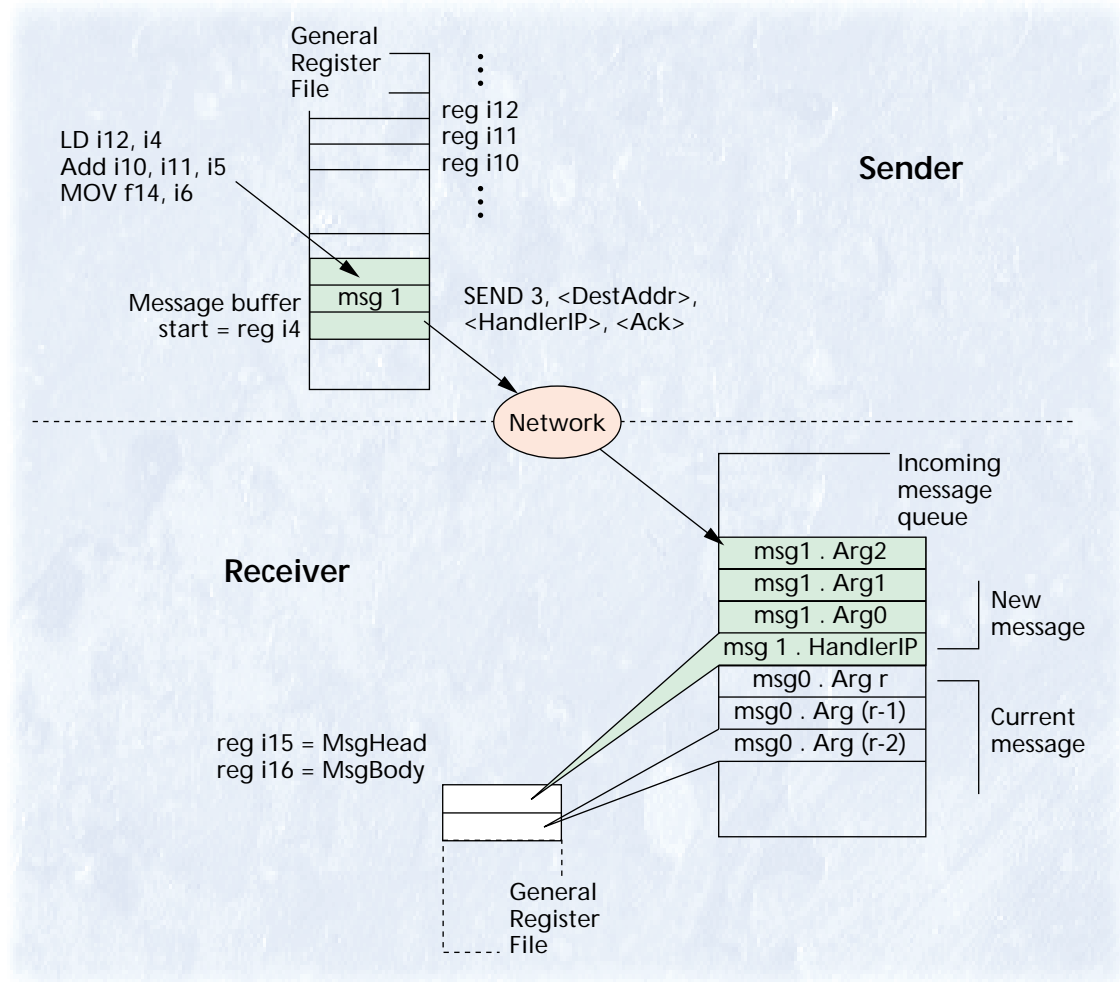
In examining the impact of design choices on message interface performance, we find that the dispatch mechanism is critical. A message can be dispatched up to 18 times faster by reserving a hardware thread context for message reception instead of an interrupt-driven interface. The mapping decision is also important, with integrated register-mapped interfaces as much as 3.5 times more efficient than conventional systems. With fine-grain messages, atomicity does not have a very significant impact on performance. However, in a processor containing multiple hardware thread slots, the atomicity provided by buffered interfaces is important for protection and preventing starvation.

An enduring design challenge is to select a mix of mechanisms that complement one another, enhance protection, improve raw performance, and reduce overhead.

THE MIT M-MACHINE

Designed to exploit parallelism, the MIT M-Machine consists of an array of Multi-ALU Processor (MAP) nodes connected to each other in a two-dimensional mesh. The MAP chip contains three execution clusters, a two-bank unified cache, and an external memory interface. An on-chip NI and a two-dimensional router allow multiple MAP chips to be connected in the M-Machine. In this system, clusters make memory requests to the interleaved cache banks over the 3×2 M-Switch crossbar, which connects the three clusters to the two interleaved cache banks. The 7×3 C-Switch crossbar provides intercluster communication, returns data from the memory system, and connects the clusters to two outgoing message queues.

Figure 1. The architecture of the MIT M-Machine message interfaces.



Each of the three execution clusters is a 64-bit, three-issue, pipelined processor that has two integer ALUs, a floating-point ALU, register files, and a 4-Kbyte instruction cache. Due to area constraints, only one FPU is implemented in the MAP prototype chip, although the simulation studies performed here assume an FPU for each of the three clusters.

Each cluster implements cycle-by-cycle multi-threading, with the register file and pipeline registers replicated for five independent thread slots. Each thread includes 14 integer registers, 15 floating-point registers, and 16 Boolean condition-code (CC) registers. Instructions from the threads are interleaved over the execution units on a cycle-by-cycle basis with no pipeline stalls when switching between threads. A synchronization pipeline stage selects the thread to issue based upon resource availability and data dependency, using a scoreboard to keep track of the validity of each register.

M-MACHINE MESSAGE INTERFACE ARCHITECTURE

The M-Machine maps its message interfaces—illustrated in Figure 1—into the processor’s general register name space and pairs a buffered, atomic injection interface with a streaming extraction interface. This system dispatches messages asynchronously within a jump delay (of three cycles) upon arrival.

Injection

As shown in Figure 1, a user thread first assembles the message body (which can be up to 10 words in length) in either its integer or its floating-point register files, starting at register i4 or f4. A nonblocking SEND instruction then atomically injects the message into the network: SEND <length>, <DestAddr>, <HandlerIP>, <Ack>. A virtual memory pointer, *DestAddr*, specifies the destination. During injection a small hardware cache—known as the *global translation look-aside buffer* (GTLB)—translates *DestAddr* into physical routing information, which directs the message through the network.

The action at the receiving end is specified by *HandlerIP*, which is an instruction pointer to a message handler routine. The M-Machine requires *DestAddr* and *HandlerIP* to be unforgeable pointers,⁷ and aborts the SEND instruction with a protection-violation exception if either is found to be invalid. *Ack* specifies a condition register to be validated after the network controller has retrieved the message from the register file. As soon as the system issues the SEND instruction, the program can proceed with further computation as long as it avoids contaminating the message registers or getting swapped out before *Ack* is validated.

Extraction

The M-Machine reserves two independent thread

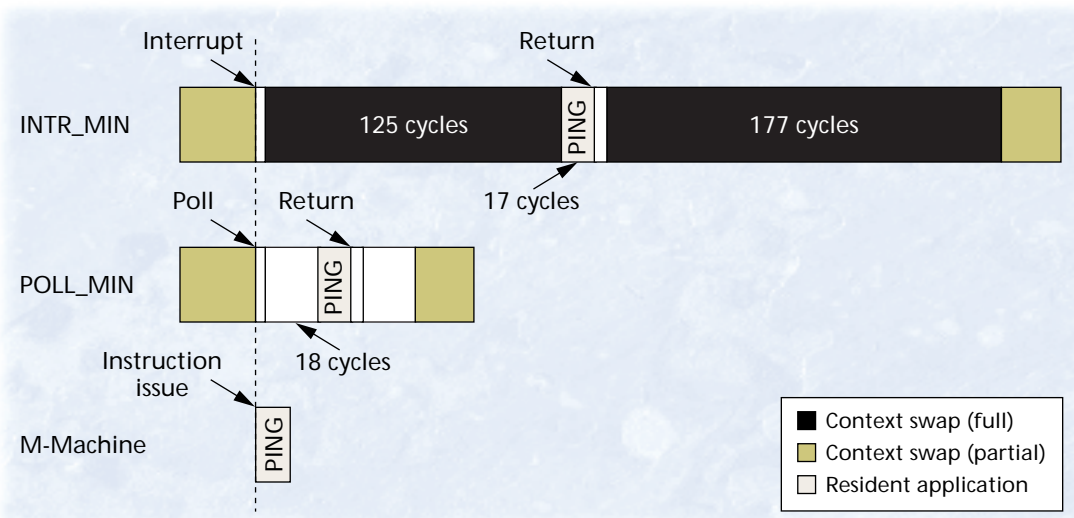


Figure 2. The latency components at the receiving end of the ping test.

slots for message reception, one for each message priority, and maps integer registers `i14 (MsgHead)` and `i15 (MsgBody)` in each of these thread slots to a corresponding incoming message queue, as shown in Figure 1. Whenever `MsgHead` is read, the network hardware returns the handler IP of the next message, discarding any remaining words from the current message.

Reading `MsgBody` returns the next word in the current message instead. In either case, the consumed word is also popped from the queue. Thus, a sequence of reads to `MsgBody` returns the subsequent words in a message. After the system consumes a message tail, the NI unit pads further reads to `MsgBody` with dummy values until the next message is scrolled in by a read to `MsgHead`. Both `MsgHead` and `MsgBody` can be used directly as the source operand in any regular instruction.

Dispatch

For `MsgHead` and `MsgBody`, the system maps the corresponding scoreboard bits to the presence of a new message and the availability of the next word in the current message. Consequently, an instruction relying on these registers does not issue until the corresponding message word is available. This allows a message dispatcher installed in the reserved thread slot to wait for message arrival without consuming any execution resources, yet still remain able to activate immediately when the first message word arrives.

PERFORMANCE EVALUATION

We evaluated the performance impact of design choices in primitive messaging mechanisms using three benchmarks:

- *Ping*, which measures request-response time between two nodes;
- *Remote Procedure Call (RPC)*, which measures the time it takes to send an eight-argument message to spawn a new remote thread; and
- *Distribute*, which measures the time it takes to send eight RPC messages to eight different nodes.

To gauge the efficiency of doing block transfers, we

also used the *Blockwrite* benchmark, which measures the speed of a 1,024-word remote-memory transfer in packets of 10-word messages.

We explored the design space by running the benchmarks over system models with a streamed or buffered protocol, register-based or memory-based interface map, using interrupt-driven, polled, and M-Machine-style dedicated-thread dispatch mechanisms. Since we are experimenting with various combinations of mechanisms, these models may not correspond exactly to existing architectures. However, familiar points of reference include the CM-5 (memory-mapped streaming injection and extraction),¹ the J-Machine (register-mapped streaming injection and extraction),² and Shrimp (memory-mapped buffered injection and extraction).⁵

For each experiment, we measured the latency from message creation to the last message-driven event. We also measured processor occupancy as the sum of all cycles used by message-related operations, including message creation and handling. We conducted all experiments on `msim`, a C-level simulator used for verification of the M-Machine implementation. `Msim` is accurate to within 10 percent of actual cycle times and is augmented to simulate both register-buffered injection and register-mapped streaming injection.

Dispatch mechanisms

Figure 2 shows the receiving-end latency components for ping with different dispatch mechanisms. Upon an interrupt (`INTR_MIN`), swapping the entire thread context (32 registers in our benchmarks) causes an overhead that is nearly 18 times the actual ping service time. The polling mechanism (`POLL_MIN`) is less costly since the polling program knows to save only the known live registers. However, the resulting ping response is still more than three times slower than the M-Machine architecture.

Figure 3 illustrates similar latency trends for each benchmark. `POLL_MIN` indicates the best case scenario, where the message arrives exactly when the polling takes place, while `POLL_MAX` represents the worst-case results, when message arrival misses the

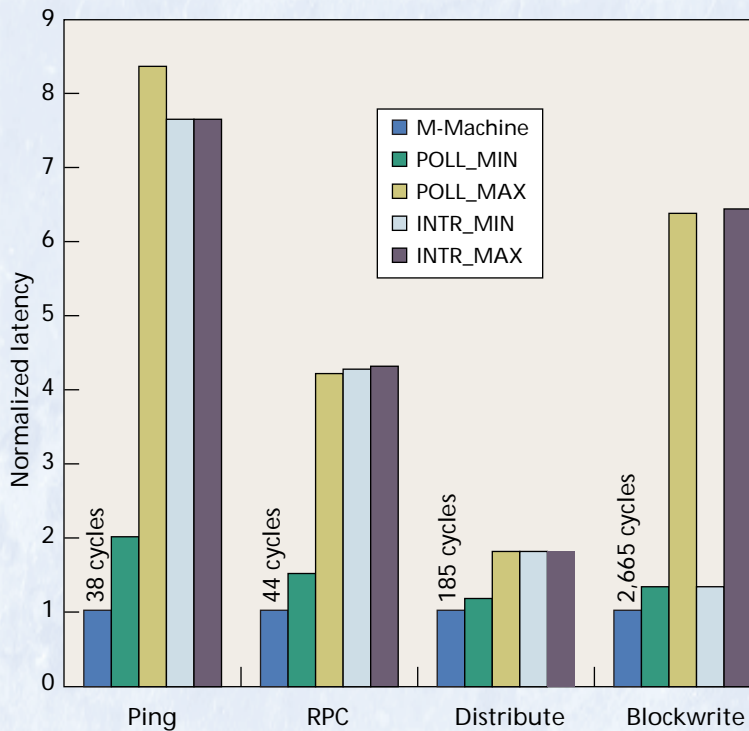


Figure 3. Normal latencies for each of the four benchmark tests.

poll by one cycle. In both the polling and the interrupt-driven models, our handlers check for new messages before returning to the displaced application. `INTR_MIN` and `INTR_MAX` represent the best-case and worst-case scenarios where subsequent message arrivals happen to hit and miss this check, respectively.

Naturally, the difference between `INTR_MIN` and `INTR_MAX` is only relevant when multiple messages are received consecutively in the benchmark, as in Blockwrite. However, the good `INTR_MIN` result for Blockwrite is deceptive, because it shuts out the displaced application for an extended period of time—until all 103 messages in Blockwrite have been received.

Mapping

The address setup overhead in memory-mapped interfaces causes as much as 1.6 times the processor occupancy of the M-Machine. Those extra instructions, together with the latency incurred as each message word traverses the on-chip memory hierarchy to reach the pins, also make these interfaces up to 3.5 times slower than the corresponding integrated mechanisms.

In register-mapped interfaces, performance is degraded if message words must be explicitly copied to and from a register name-space separate from the general register file. In the M-Machine extraction interface, both `MsgHead` and `MsgBody` can be used directly as instruction operands, giving the interface a slight advantage over others in which the network-mapped register only supports the copy instruction. While conventional integrated interfaces could be used in the M-Machine to achieve competitive latency results, they would have to use multiple function units to overlap extraction with message handling and pay for it with higher processor occupancy.

Block transfers often motivate incorporating Direct Memory Access (DMA) engines into a system. However, traditional DMA interfaces are advantageous only for large transfers because they incur thousands of cycles of overhead in system calls. Although the per-transfer cost can be reduced to several hundred cycles via user-level DMA mechanisms, an expensive system call is still required to set up the sender-receiver DMA buffer mapping. Therefore, for moving a moderate amount of data, a DMA system is not necessarily faster than the software-packetized M-Machine model, which already uses roughly 38 percent of the network bandwidth and avoids cache pollution with uncached memory load instructions. However, for very large transfers, a DMA system does incur lower processor occupancy.

Buffering versus streaming

Streaming injection interfaces benefit from being able to overlap message assembly and injection time, yielding latency savings proportional to the sum of all delays during message creation, including cache misses. Figure 4 contrasts the latency components in RPC for the three tightly integrated injection interfaces. By targeting message-generating instructions directly into the message-buffer-mapped registers, the M-Machine avoids explicitly copying messages into the message buffer, as is necessary in the more conventional `INJ_REG_BUF` call.

Figure 4 also shows that overlapping message creation with injection makes the streaming interface even faster. But that is not always the case. For example, with very short messages, the memory-mapped streaming channel setup cost sometimes dominates the per-word buffering overhead. A streaming interface is also unable to exploit message reuse. When message assembly time is amortized over several messages, as in Distribute, `INJ_REG_STR` lags behind M-Machine and `INJ_REG_BUF`. In Blockwrite, the latency results are similar among the three designs due to aggressive software-pipelining compensating for buffering delays.

With its relatively small register files, the M-Machine faces register pressure in Distribute, where four arguments are regenerated for each message. With the message occupying 10 registers, too few registers are left to overlap that computation with injection. Although messages can be pipelined from the integer and floating-point register files in the M-Machine, this capability is limited in Distribute because the regeneration phase happens to use some instructions that are specific to the integer unit. In architectures with larger register files, this problem can conceivably be alleviated by pipelining messages (from different portions of each register file) using a simple modification to allow the message buffer to be placed anywhere within the registers.

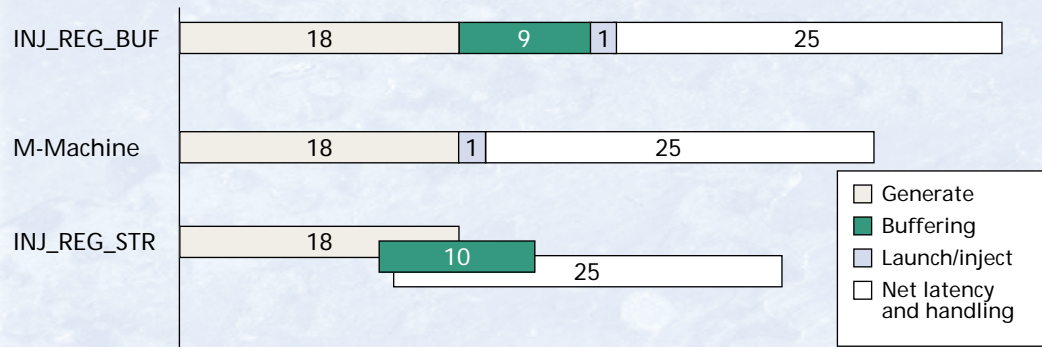


Figure 4. Latency components in Remote Procedure Call (RPC) message injection.

Latency and architecture

Figure 5 summarizes the end-to-end latency for ping when gradually switching from a traditional memory-mapped, buffered, interrupt-driven message interface to the M-Machine architecture. The most significant latency reduction comes from eliminating the context-swap upon dispatch (which can amount to roughly 60 percent in time savings). The various memory-mapped interface options do not differ much in performance. Substituting the M-Machine register-based mapping produces another 30 percent improvement. The fast address translation mechanism in the GTLB provides the remaining latency reduction. Taken in sum, the M-Machine message architecture delivers up to an order of magnitude performance improvement, even before considering the system-call layer often required by traditional messaging systems.

MULTITHREADED MESSAGING

The message interface is subject to conflicts when multiple threads attempt to access it concurrently. It is the system's responsibility to guarantee noninterference between concurrent threads by granting exclusive use of the shared messaging facilities according to need, while at the same time preventing any thread from monopolizing resources and starving other threads. The message system should also seamlessly extend its protection system beyond node boundaries. We describe in this section how the M-Machine efficiently supports these needs without slow software semaphores or authentication systems.

Resource sharing

Figure 6 shows the possible configurations for resource sharing in a multithreaded message system. Resources can be shared on the basis of an open-ended exclusive allocation, fixed time-slicing, or a "bounded time" lease. In Figure 6a, a streaming injection interface exclusively allocates virtual channels from a resource pool to threads on demand, which allows multiple logical connections to be open concurrently. Although the traffic from these virtual channels can be multiplexed onto the network port on a time-slice basis, each exclusively allocated virtual channel cannot be reused until it is voluntarily returned to the resource pool by the user. This allows programs to cause starvation easily by exhausting the shared

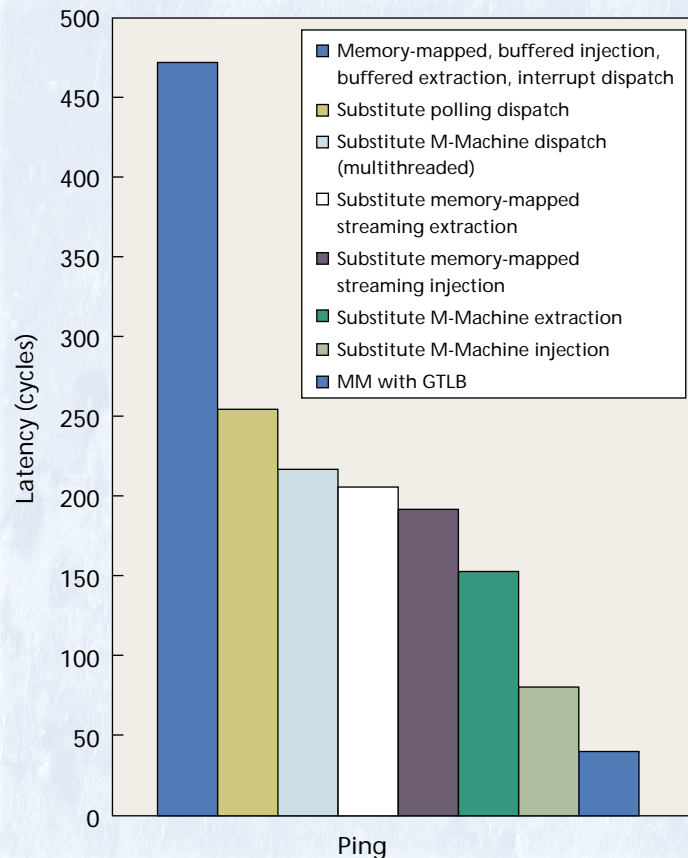


Figure 5. End-to-end latency for ping when gradually switching from a traditional memory-mapped, buffered message interface to the M-Machine architecture.

resources, intentionally or otherwise.

This inability to reclaim shared resources is a fundamental problem of open-ended exclusive allocation schemes. In Figure 6b, a buffered injection interface also multiplexes a pool of message composition buffers onto the network port. Since each message is atomically injected, the network port needs to be connected to a buffer only for a bounded duration. An appropriate fair arbitration model can thus be used here to prevent starvation. But a potential danger still exists, since the buffer pool itself is shared among threads based on an open-ended allocation scheme, which may allow a thread to monopolize the buffers.

Figure 6c shows a model that eliminates the exclusive allocation step by hardwiring a message buffer to each hardware thread slot. This technique removes dynamic buffer allocation overhead and eliminates

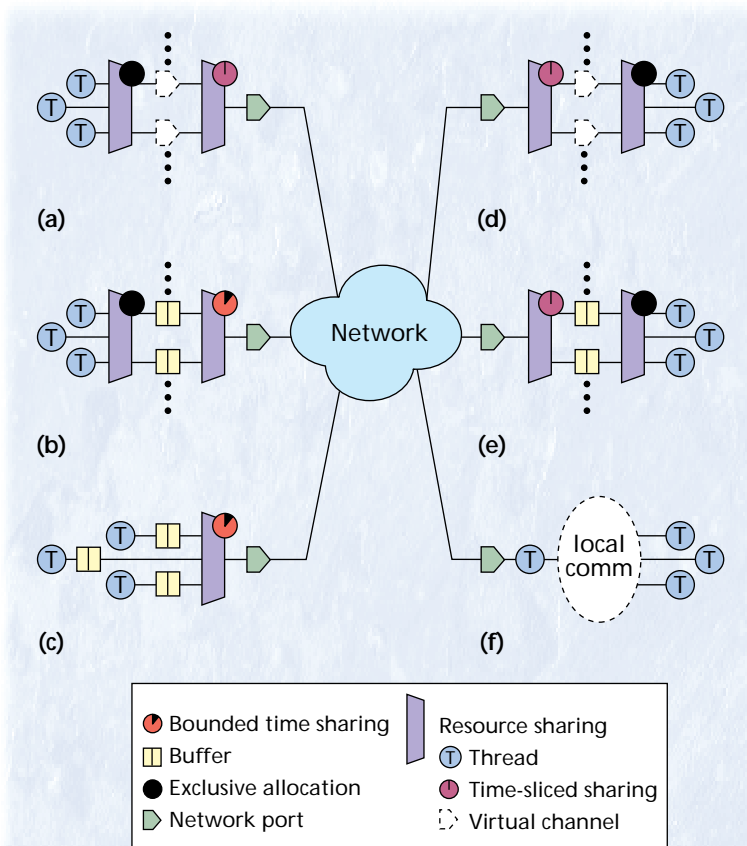


Figure 6. Resource sharing in a multi-threaded message system using (a) shared injection channels, (b) shared injection buffers, (c) dedicated injection buffers, (d) shared extraction channels, (e) shared extraction buffers, and (f) a message handler.

the risk of starvation since the execution of thread slots is already fairly arbitrated in hardware. The message buffer then becomes an integral part of the thread context. To avoid the high latency of main memory messaging operations, each dedicated buffer can conceivably be implemented in special on-chip memory, but this technique would waste valuable chip area. By buffering in existing register files, the M-Machine prevents starvation without wasting chip area on buffers that are often idling. In addition, this type of mapping is naturally scalable as the number of thread slots or processing clusters grow on the chip.

We face a similar set of trade-offs in the design of a message extraction interface. Figure 6d shows an interface that shares a pool of extraction channels among a set of user threads, while Figure 6e shows the corresponding shared buffer interface. In both models, since user threads are granted direct access to the critical network components, incoming messages are subject to blocking when the shared resources are not relinquished promptly.

As shown in Figure 6f, the M-Machine uses a more robust and flexible alternative extraction interface, which is closely related to the Active Message communication model.⁸ Instead of being received by a particular thread, each message designates a handler routine to be invoked at the destination. The handler is given exclusive access to the extraction interface so that it can flexibly react to the message. To avoid starvation and deadlocks, conventional Active Message-like implementations count on the handlers to

complete their tasks and return control to the system quickly. The M-Machine, however, provides an enforcement mechanism—through its protection system described below—so that only safe, trusted handlers are accessible to the user.

Protection

The M-Machine extends protection domains across multiple nodes by restricting both the set of processors to which a thread can send a message and the handlers that can be invoked at the destination. In the M-Machine, all of the memory resides in a single global virtual address space. Protection domains are implemented not by different address spaces, but by using the segmentation and capabilities of guarded pointers.⁷ In a SEND instruction, the destination must be specified with an unforgeable virtual address pointer. Since the GTLB transparently maps these addresses to physical nodes, a thread can only communicate with nodes within its own protection domain.

In order to prevent a user from invoking an ill-behaving message handler, the guarded pointer system is also used to implement *trusted handlers* in the M-Machine. A trusted handler is a user or system routine certified to be safe. A trusted handler never blocks indefinitely and is guaranteed not to cause any unrecoverable errors. Certification may be done through careful human inspection or compiler analysis. In the M-Machine, the SEND instruction requires that `HandlerIP`—which specifies the invoked handler—be an instruction pointer of type `Execute-Message`. An `Execute-Message` pointer cannot normally be executed or modified by user-level programs. However, as the message is injected into the network, the hardware transparently converts and transmits `HandlerIP` as an executable instruction pointer instead. Therefore, trusted handlers can only be invoked via the message system. By selectively making `Execute-Message` pointers available to each thread, the system can regulate the remote operations accessible to the thread while ensuring that only well-behaving message handlers are given access to the extraction interface.

The SEND instruction causes an exception and aborts the message if either `DestAddr` or `HandlerIP` is the wrong type of guarded pointer. No authentication is required at the destination. To enhance protection even more, the M-Machine discards the remaining words of the current message when `MsgHead` is read and pads the end of each message with null values to thwart a user's attempts at confusing the handler with a message of unexpected length.

A fast, low-overhead message subsystem is essential for efficient multicomputing. As we've explained, a dedicated-thread mechanism can cut dispatch latency by as much as 18 times the latency

of an interrupt-driven interface. Mapping the message interface to memory—instead of integrating it with the processor—costs up to 3.5 times more in end-to-end latency. For short messages, however, buffered and streaming models do not substantially differ. Raw performance alone, however, is by no means sufficient. To meet the challenges and exploit the opportunities presented by emerging multithreaded processor architectures, low overhead mechanisms for protection against message corruption, interception, and starvation must be integral to the message system design, as they are in the M-Machine.

With increasing demand for computing power, multiprocessing computers will become more common in the future. In these systems, the growing discrepancy between processor and memory technologies will cause tightly integrated message interfaces to be essential for achieving the necessary efficiency, which is especially important in light of the growing interest in software-distributed, shared-memory systems.

Increasing effective chip area has also enabled novel architectures that exploit on-chip parallelism. When incorporated into networks of workstations and multicomputers, these emerging architectures will provide low-cost, high-performance computing. However, with multiple processors and thread slots on each chip, such systems will encounter many protection, resource-allocation, and starvation issues. The simple messaging mechanisms described here can help provide a solution to these challenges, as they have in the M-Machine. ❖

Acknowledgments

The research described in this article was supported by the Defense Advanced Research Projects Agency under ARPA order 8272 and monitored by the Air Force Electronic Systems Division under contract F19628-92-C-0045. We thank the anonymous reviewers for their valuable feedback.

References

1. C. Leiserson et al., "The Network Architecture of the Connection Machine CM-5," *Proc. Symp. Parallel Algorithms and Architectures*, ACM Press, New York, 1992, pp. 272-285.
2. W.J. Dally et al., "The J-Machine: A Fine-Grain Concurrent Computer," *Proc. Information Processing 89*, Elsevier Science, North Holland, 1989, pp. 1,147-1,153.
3. K. Mackenzie et al., "Exploiting Two-Case Delivery for Fast Protected Messaging," *Proc. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 231-242.
4. J. Kuskin et al., "The Stanford Flash Multicomputer," *Int'l Symp. Computer Architecture*, ACM Press, New York, 1994, pp. 302-313.
5. M.A. Blumrich et al., "Protected User-Level DMA for the Shrimp Network Interface," *Proc. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1996.
6. M. Fillo et al., "The M-Machine Multicomputer," *Proc. 28th Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 104-114.
7. N. Carter, S. Keckler, and W. Dally, "Hardware Support for Fast Capability-Based Addressing," *Architectural Support for Programming Languages and Operating Systems*, IEEE CS Press, Los Alamitos, Calif., 1994.
8. T. von Eicken et al., "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 256-266.

Whay Sing Lee is the communication system architect for the MIT M-Machine. His research interests include computer architecture, multicomputer networks, and fault-tolerant computing. Lee received an MS in computer science from the Massachusetts Institute of Technology, where he is also a PhD candidate.

William J. Dally is a professor of electrical engineering and computer science at Stanford University, where he leads projects on high-speed signaling, multiprocessor architecture, and graphics architecture. Dally received an MS in electrical engineering from Stanford University and a PhD in computer science from the California Institute of Technology.

Stephen W. Keckler is an assistant professor of computer science at the University of Texas at Austin. His research interests include computer architecture, parallel and embedded processors, VLSI design, and the relationship between technology and computer systems development. Keckler received an MS and a PhD in computer science from the Massachusetts Institute of Technology.

Nicholas P. Carter is a PhD candidate at the Massachusetts Institute of Technology, where he received an MS in computer science. His research interests include shared-memory computer systems, processor architectures to minimize communication, and circuits for on-chip signaling.

Andrew Chang is a PhD student in the Concurrent VLSI Architecture group at Stanford University. His research interests include VLSI, CAD, computer architecture, and circuit design. Chang received an MS from the Massachusetts Institute of Technology.

Contact the authors at {wslee, billd, skeckler, achang, npcarter}@cva.stanford.edu.