Efficient, Protected Message Interface in the MIT M-Machine

Whay Sing Lee * William J. Dally Stephen W. Keckler

wslee@cva.stanford.edu billd@cva.stanford.edu skeckler@cva.stanford.edu

Nicholas P. Carter Andrew Chang

npcarter@cva.stanford.edu achang@cva.stanford.edu

* Massachusetts Institute of Technology

Computer Systems Laboratory, Stanford University

June 17, 1998

Abstract

We present a user-level message interface that provides high performance and very low processor overhead. In this system, messages are launched from within the user's general register file, and received in a hardware queue mapped to a general register. A message handler is started within the latency of a *jump* instruction upon arrival of the first message word, up to 18× faster than conventional interrupt-driven interfaces. These tightly-integrated mechanisms feature end-to-end latencies as low as nearly one quarter that of memory-mapped interfaces. Copying is eliminated in these mechanisms, reducing processor occupancy by up to one-third when compared with other integrated register-mapped systems. The interface also features a low overhead, robust protection model using *virtually-addressed* message destinations and certified *trusted handlers.* Both are specified with unforgeable pointers, enabling fine-grain control over a user thread's accessible domain as well as its permitted remote operations. We discuss the application of this system in the framework of the multithreaded MIT M-Machine, and show that unlike other approaches, it is able to provide protection and avoid starvation while maintaining high efficiency.

1 Introduction

The message interface defines the abstraction layer between the network hardware and the software. In traditional message interfaces, high latency and processor occupancy (unavailability of processing resources for productive computation during message operations) inhibit efficient exploitation of large-scale parallelism. Recent designs address this problem by removing OS layers from the interface [4, 8, 10, 11, 12]. Nonetheless, the remaining overhead is still large. For instance, the initialization cost alone for a round-trip null message is >330 cycles in the SHRIMP [12]. To amortize communication overhead of 100s-of-cycles, programmers are motivated to use messages that are 100s to 1000s of words or larger in size. Consequently, threads require >1000s of cycles in run-length between communications, precluding many parallelization opportunities, such as inner-loop level concurrency.

When multiple threads of control are incorporated onto each node, this overhead is exacerbated if primitive support for fair and protected resource allocation is lacking. However, much of the communication overhead can be removed by carefully making complementary design choices in primitive messaging mechanisms, to facilitate messages as short as several words in size, and enable fine-grain parallelization. For instance, a *complete* round-trip null message takes only 38 cycles in the M-Machine.

The design space for messaging mechanisms can be described in terms of three aspects: (a) mapping, (b) atomicity, and (c) dispatch. The *mapping* defines how the network interface hardware is presented to the software. Memory-mapped interfaces access the network state through specific addresses as if it is part of the memory, while instruction/register-mapped interfaces integrate tightly with the processor, injecting and extracting messages with special instructions or by reading and writing special device registers. *Atomicity* refers to the granularity of message transfer. A *streaming* interface allows the message head to "wormhole" piece-meal through the system without waiting for its tail. Conversely, a *buffered* interface requires each message to be completely stored into a buffer before it is injected in an un-interruptible fashion, or received as an atomic unit. The *dispatch* mechanism determines how the processor reacts to message arrivals. Conventionally, an interrupt mechanism asynchronously displaces the currently-executing program with an interrupt handler upon message arrival, while a polling system periodically checks for message arrival in software.

In examining the impact of design choices on message interface performance, we find that the dispatch mechanism is critical. A message is dispatched up to 18 times faster when a hardware thread context is reserved for message reception, compared to an interrupt-driven interface. The mapping decision is also important, with memory-mapped interfaces being as much as 3.5× more efficient. With fine-grain messages, the atomicity choice does not have very significant performance impacts. However, in a processor containing multiple threads of control, the atomicity of buffered interfaces is important for protection and starvation prevention.

In the next section, we describe the MIT M-Machine and its message system, which incorporates a mix of mechanisms that boost performance without compromising protection. Performance evaluation results are presented in Section 3. In Section 4, we discuss additional issues when the message interface is integrated into multithreaded systems.

2 The MIT M-Machine

The MIT M-Machine [1] is an experimental multicomputer designed to exploit parallelism with a wide range of granularity. It consists of an array of Multi-ALU Processor (MAP) nodes connected in a 2-D network supporting two message priorities. The network links have a 1-word/cycle raw bandwidth.

2.1 The MAP Chip

Figure 1 shows a preliminary plot of the MAP chip containing three execution clusters, a two-banked, unified cache, and an external memory interface. An on-chip network interface and two-dimensional router allow multiple MAP chips to be connected into a multicomputer. Clusters make memory requests to the interleaved cache banks over the 3 × 2 M–Switch crossbar. The 7 × 3 C–Switch crossbar provides inter-cluster communication, returns data from the memory system, and connects the clusters to two outgoing message queues. Each of the three clusters is a 64-bit, three-issue, pipelined processor with two integer ALUs, a floating–point ALU, associated register files, and a 4KB instruction cache[1].

[1] In the silicon implementation of the MAP architecture, only cluster 0 has a floating–point unit, due to chip area constraints. The simulation studies performed in this paper include floating–point units for each of the three clusters.

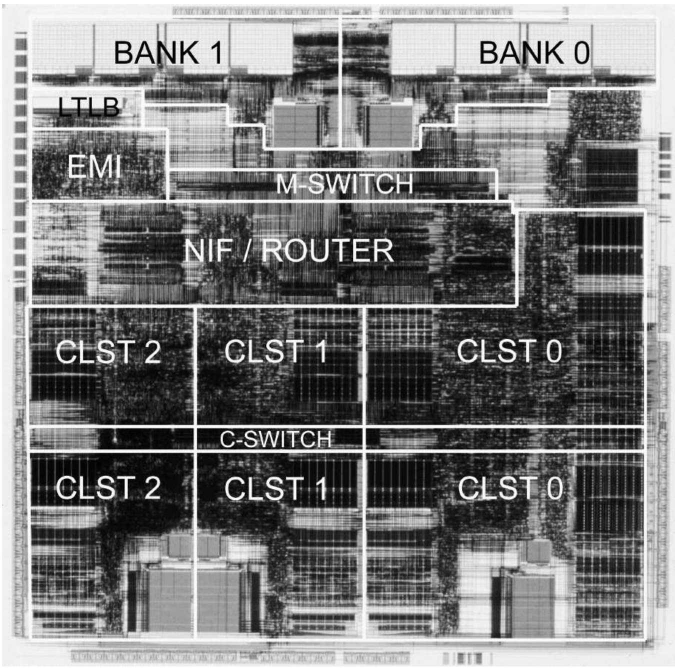

Figure 1: Preliminary plot of the MAP chip in $0.7um$ (drawn gate-length) CMOS technology, measuring $18mm$ on a side and containing approximately 5 million transistors.

Each cluster implements cycle-by-cycle multithreading, with the register files and pipeline registers of the top stages of the pipeline replicated for five independent *thread slots*. Each thread has 14 integer registers, 15 floating–point registers, and also 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.

2.2 M-Machine Message Interface Architecture

The M-Machine message interfaces, illustrated logically in Figure 2, are completely mapped into the processor's general register name space. A buffered, atomic injection interface is paired with a streaming extraction interface. Messages are dispatched asynchronously upon arrival within a *jump* delay (3 cycles).

INJECTION A user thread first assembles the message body, up to 10 words in length, in either its integer or floating-point register-files, starting at register *i4* or *f4*, respectively (Figure 2). A non-blocking *SEND* instruction then atomically injects the message into the network:

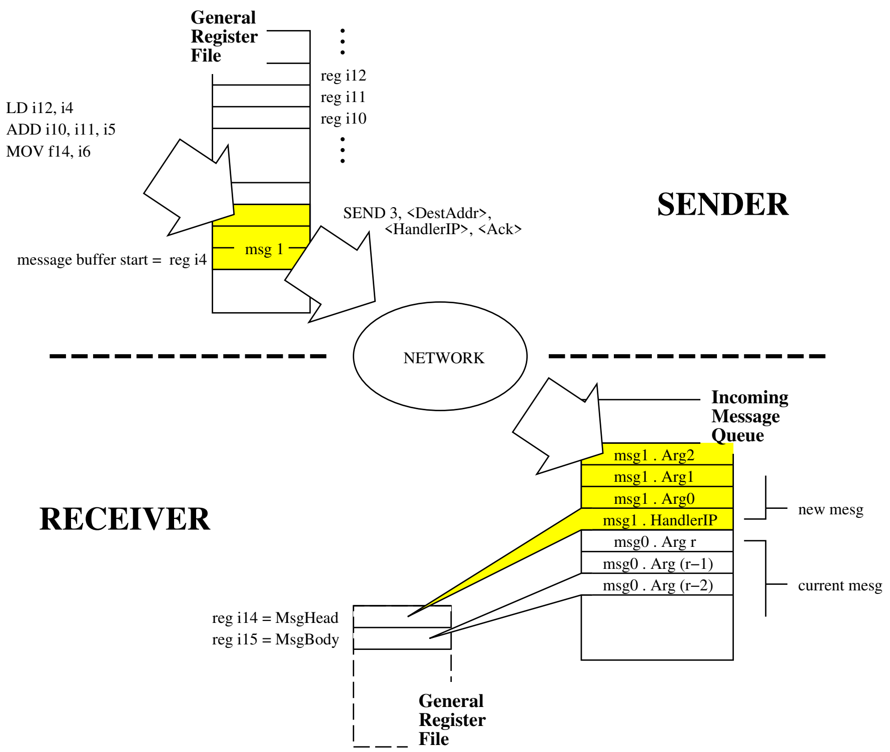

Figure 2: M-Machine Message Interfaces

```
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 [2], and aborts the SEND instruction with a protection violation exception if either is found to be invalid. *Ack* specifies a condition (CC) register to be validated after the network controller has retrieved the message words from the register file. As soon as the SEND instruction is issued, 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 slots for message reception, one for each message priority. Integer registers *i14* (*MsgHead*) and *i15* (*MsgBody*) in each of these thread slots are mapped to its corresponding incoming message queue (Figure 2). Whenever *MsgHead* is read, the network hardware returns the *HandlerIP* 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 tail of a message is consumed, the network interface unit pads further reads to *MsgBody* with a dummy value, 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 corresponding register scoreboard presence bits are mapped to the presence of a new message and the availability of the next word in the current message, respectively. Consequently, an instruction sourcing 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 remain able to activate immediately when the first message word arrives. The example below shows the initial message dispatcher and a remote-write message handler for storing a word into memory. Note the direct use of *MsgBody* as the operand to *store* (`st`).

```
/* Incoming message format:                                                 */
/*      MsgHead            = pointer to _remote_write_handler               */
/*      MsgBody (1st word) = target write address                           */
/*      MsgBody (2nd word) = data word                                      */

_msg_dispatcher:                          /* initially stalls on MsgHead     */
        instr   ialu jmp MsgHead;         /* jump to _remote_write_handler_  */

_remote_write_handler:
        instr   ialu mov MsgBody, i6;     /* target write address            */
        instr   memu st MsgBody, i6;      /* data to store in target address */
        instr   ialu jmp MsgHead;         /* done. dispatch next message     */
```

3 Performance Evaluation

The performance impact of design choices in primitive messaging mechanisms is evaluated using 3 fine-grain micro-benchmarks: PING, RPC, and DISTRIBUTE. PING is a simple two-node request-response program. The RPC benchmark sends an 8-argument message to remotely spawn a new thread. In DISTRIBUTE, 8 RPC messages are sent to 8 different nodes, where four of the arguments are identical across all 8 messages and four others are different. To gauge the efficiency in doing block transfers, we also included the BLOCKWRITE benchmark, which performs a 1024-Word remote-memory transfer, packetized into 10-word messages.

We explore the design space by running the benchmarks over system models with a streaming or buffered protocol, register-based or memory-based interface mapping, and message dispatch using a interrupt-driven, polling, or M-Machine style dedicated-thread mechanism. 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 [4] (memory-mapped streaming injection and extraction), the *T [5] (register-mapped buffered injection and extraction), the J-Machine [8] (register-mapped streaming injection[2] and extraction), and SHRIMP [12] (memory-mapped buffered injection and extraction).

For each experiment, we measure the latency from message-creation to the last message-driven event (*eg.* execution of first instruction in the last spawned thread in DISTRIBUTE), and processor occupancy as the sum of all function-unit-cycles used by message-related operations, including message-creation and message-handling. All experiments are conducted on *msim*, a C level simulator used for verification of the M-Machine RTL model. The simulator is cycle-accurate to within 10%, and is augmented to simulate both register-buffered injection and register-mapped streaming injection. Dummy instructions are added to the benchmarks to account for message copying operations when necessary. To approximate interrupt-driven dispatch, an *interrupt server* thread is installed in an independent hardware thread slot, to forcefully displace a dummy application upon message arrival. For polling-dispatched benchmarks, a polling frequency of 1 per 120 cycles is used.

3.1 Dispatch

Figure 3 shows the receiving-end latency components for PING under 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× the actual PING service time. The polling mechanism (POLL_MIN) is

[2]The J-Machine uses a SEND instruction to move each message word into the network. Effectively, this is equivalent to writing each message word into a special network-mapped register.

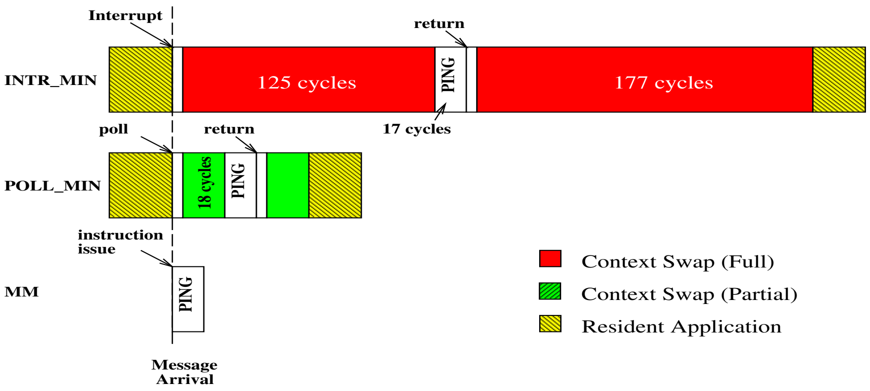

Figure 3: PING: Latency Components in Message Dispatch

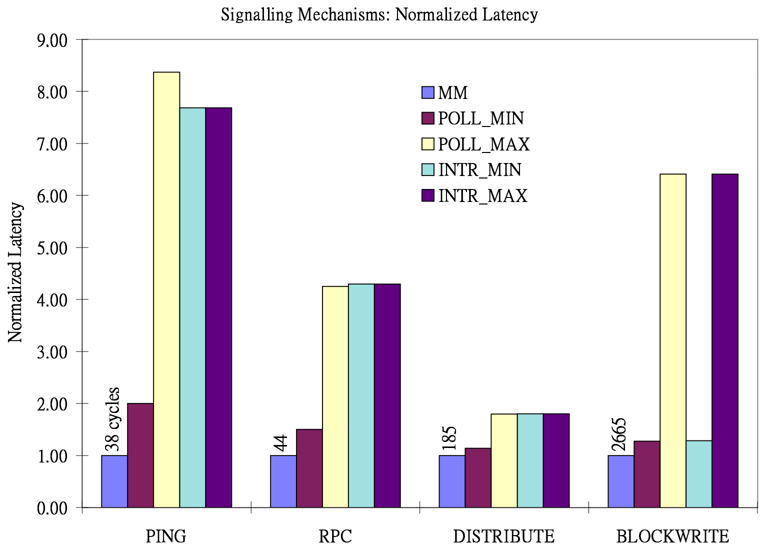

Figure 4: Dispatch Mechanisms: Normalized Latency

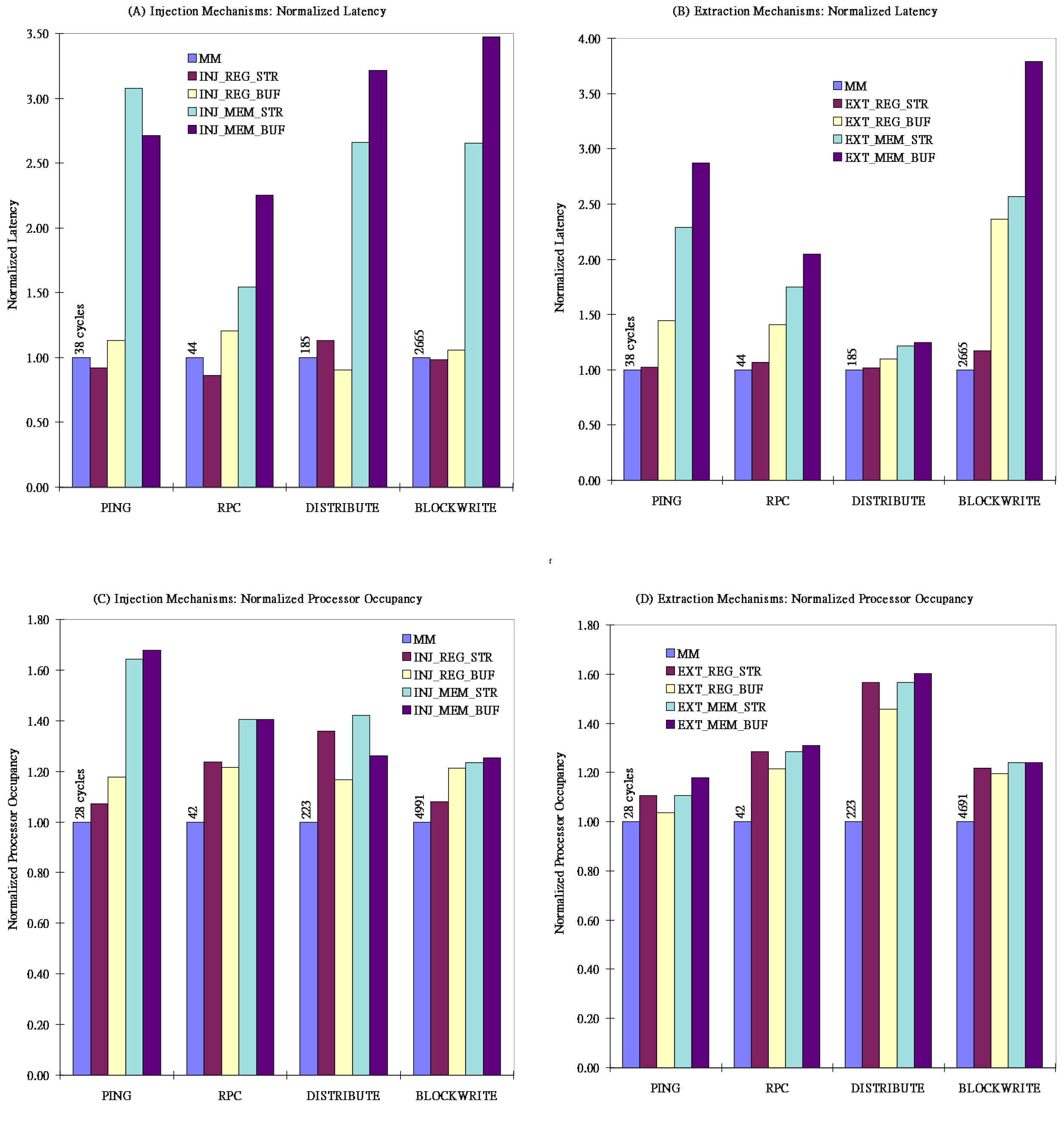

Figure 5: Injection and Extraction Mechanisms: Latency and Processor Occupancy.

less costly, since the polling program knows to save only the known live registers (6 in these benchmarks). However, the resulting PING response is still $> 3\times$ slower than of MM, the M-Machine architecture.

Figure 4 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 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 barely hit and miss this check, respectively[3]. 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, as it shuts out the displaced application for an extended period of time, until all 103 messages in BLOCKWRITE have been received.

3.2 Mapping

Figures 5A and 5B show the end-to-end latencies for the different injection and extraction mechanisms. INJ_REG_STR represents a streaming injection interface mapped to a dedicated register name space, while EXT_MEM_BUF denotes a memory-mapped, buffered extraction interface. Various other combinations of memory or register mapped, and buffered or streaming mechanisms are also shown.

Memory-mapped interfaces are up to $3.5\times$ slower than the corresponding integrated mechanisms, due to the overhead of generating addresses and securing locks prior to message operations. These extra instructions also cause as much as $1.6\times$ the processor occupancy of MM (Figures 5C and 5D). As each message word traverses the on-chip memory hierarchy to reach the pins, additional latency (10 – 30 cycles in aggressively pipelined systems as in [3]) is further incurred.

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 MM extraction interface, both *MsgHead* and *MsgBody* can be used directly as an instruction operand, giving it a slight advantage

[3]This reason also accounts for the difference between POLL_MIN and POLL_MAX in BLOCKWRITE.

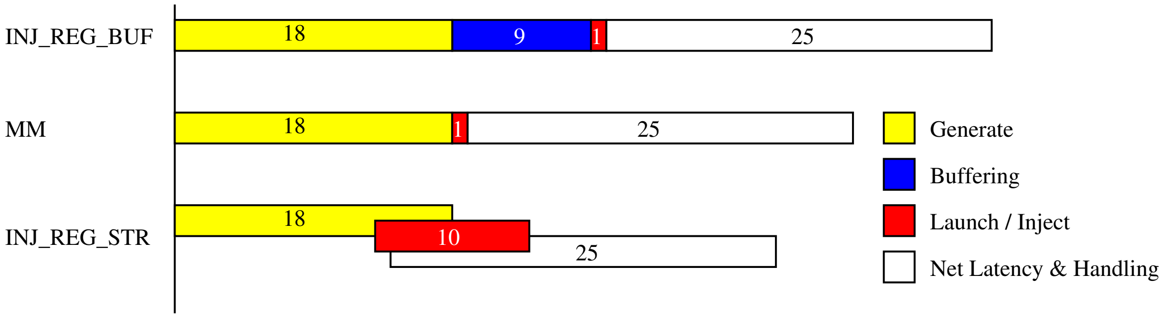

Figure 6: Latency Components in RPC Message Injection

over EXT_REG_STR, where the network-mapped register only supports the *copy* instruction. To achieve latency results competitive with MM (Figure 5B), EXT_REG_STR relies on having multiple function units for overlapping extraction with message handling, and pays for this with higher processor occupancy (Figure 5D).

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 is reduced to several hundred cycles using the *user-level* DMA mechanism in [12], an expensive system call is still required to set up the sender-receiver DMA buffer mapping. Therefore, for moving a moderate amount of data, such as 1-KWord in BLOCKWRITE, a DMA system is not necessarily faster than the software-packetized MM model, which already utilizes $\sim$ 38% 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.

3.3 Buffered versus Streaming

Streaming injection interfaces overlap message assembly with injection time, yielding latency savings proportional to the sum of all delays during message-creation, including cache misses. Figure 6 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, MM avoids the explicit copying of message words into the message buffer, as necessary in the more conventional INJ_REG_BUF. On the other hand, a streaming interface can start injecting a message as soon as the first word is ready. Nonetheless, for short messages, these effects are not significant (as low as 10% performance impact), at both the injection

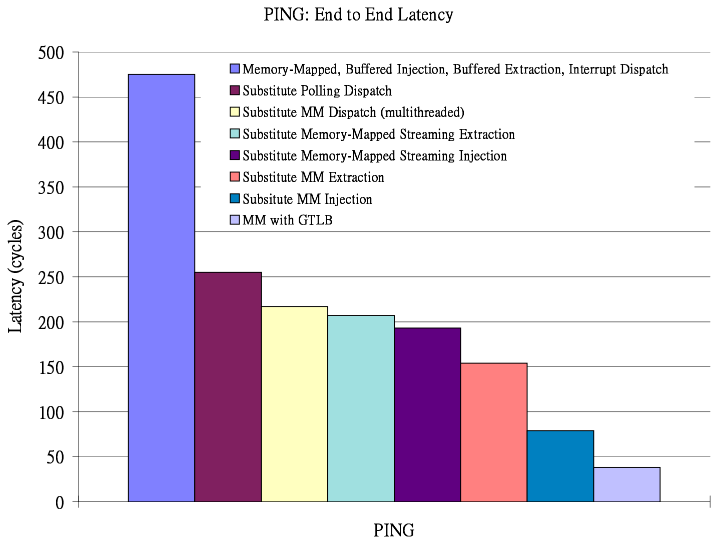

Figure 7: PING: End to End Latency, MM *vs.* Memory-Mapped Interfaces

and extraction ends.

Streaming interfaces are not always faster than their buffered counterparts. With very short messages, the memory-mapped streaming channel setup cost sometimes dominates the per-word buffering overhead, as demonstrated by PING in Figure 5A. A streaming interface is also unable to exploit message reuse. Therefore when message assembly time is amortized over several messages, as in DISTRIBUTE, INJ_REG_STR lags behind MM and INJ_REG_BUF. In BLOCKWRITE, the latency results are similar among the three tightly-integrated designs, due to aggressive software-pipelining compensating for buffering delays.

With its relatively-small register files, MM faces register pressure in DISTRIBUTE, where four arguments must be 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 (alternately) in MM, this capability is limited in DISTRIBUTE because the regeneration phase uses 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:

```
SEND <begin>, <length>, <DestAddr>, <HandlerIP>, <Ack>
```

3.4 Summary

Figure 7 summarizes the end-to-end latency for PING when gradually switching from a traditional memory-mapped, buffered message interface to the the M-Machine architecture. The most significant latency reduction comes from eliminating the context-swap upon dispatch (~60%). The various memory-mapped interface options do not differ much in performance. Substituting the M-Machine register-based mapping causes another 30% 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.

4 Multithreaded Messaging

The message interface is a valuable resource which is subject to contention, deadlock, or livelock when multiple threads of control may concurrently access the interface. The system must guarantee non-interference between concurrent threads by granting exclusive use of the shared messaging facilities according to need, while preventing any thread from causing the starvation of others by monopolizing the resources. The message system should also seamlessly extend beyond node boundaries any protection model for isolating unrelated program domains. We describe in this section how the M-Machine efficiently supports these needs without slow software semaphores or authentication systems.

4.1 Resource Sharing

Figure 8 shows the possible configurations for resource sharing in a multithreaded message system. Resources can be shared on the basis of a fixed time-slice, a bounded-time "lease", or an open-ended exclusive allocation. In Figure 8A, a streaming injection interface exclusively allocates virtual channels from a resource pool to threads on demand, allowing multiple logical connections to be open concurrently. The traffic from these virtual channels are then multiplexed onto the network port on a time-sliced basis. Each exclusively allocated virtual channel cannot be reused until it is voluntarily returned to the resource pool by the user. Therefore, programs can easily cause starvation by exhausting the shared resources, intentionally or

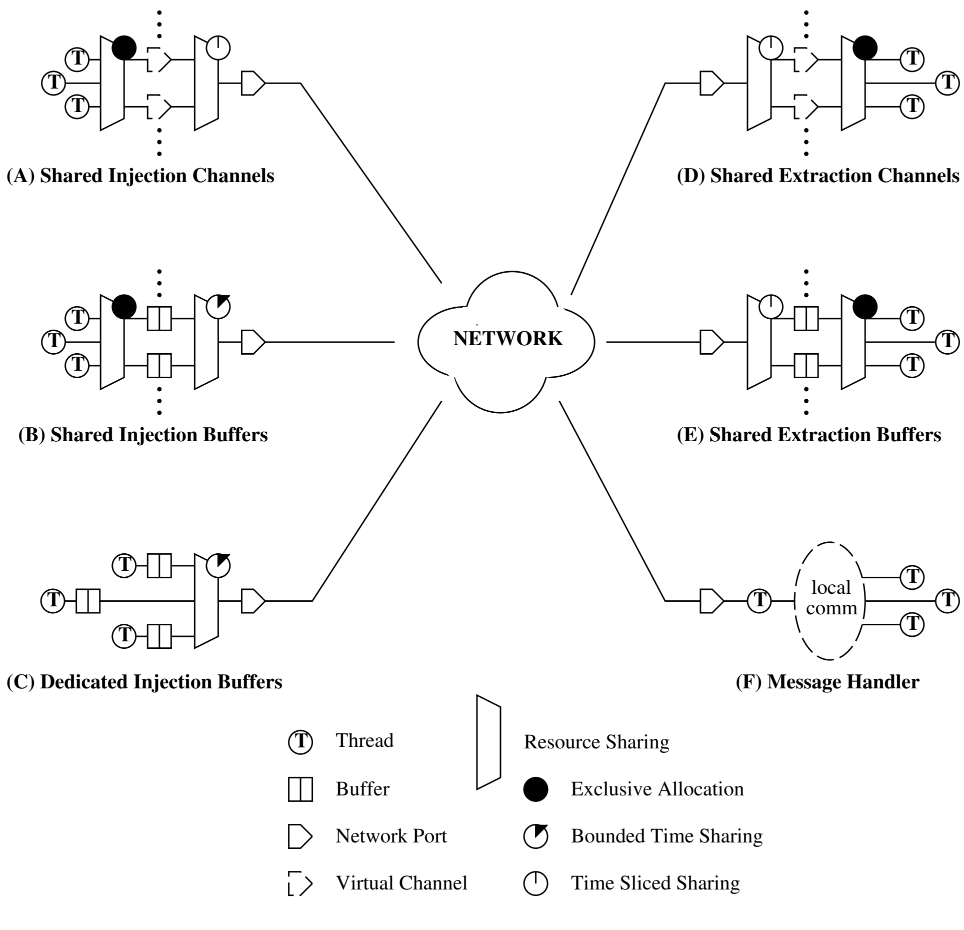

Figure 8: Multithreaded Messaging: Resource Sharing

otherwise.

This inability to reclaim shared resources is a fundamental problem with open-ended exclusive allocation schemes. In Figure 8B, a buffered injection interface also multiplexes a pool of message composition buffers onto the network port. However, because 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 to prevent starvation in this case. Nonetheless, 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 8C shows a model that eliminates the exclusive allocation step by hardwiring a message buffer to each hardware thread slot. This removes dynamic buffer allocation overhead, and the risk of starvation is eliminated since the execution of thread slots are already fairly arbitrated in hardware. Effectively, the message buffer becomes an integral part of the thread context. To avoid the high latency of main-memory based messaging operations, each dedicated buffer can conceivably be implemented in special on-chip memory. However, that would be wasteful of chip area, as message buffers tend not to have high utilization. By buffering in the register file, the M-Machine prevents starvation without wasting chip area on often-idling buffers. In addition, this mapping is naturally scalable as the number of thread slots or processing clusters grow on the chip.

A similar set of tradeoffs are faced in the design of a message extraction interface. Figure 8D shows an interface that shares a pool of extraction virtual channels among a set of user threads, while Figure 8E shows the corresponding shared buffer interface. In both models, since user threads are granted open-ended, direct access to the critical network components, incoming messages are subject to blocking when the shared resources are not relinquished promptly.

A more robust yet flexible alternative is used in the M-Machine extraction interface (Figure 8F), which is closely related to the Active Message [6] communication model. 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. Prolonged message blocking is prevented in the Active Message model by requiring handlers to always complete and return control to the system quickly, without attempting any malicious actions. In the M-Machine, this requirement is also enforced, as we describe next, using *trusted handlers*.

4.2 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 memory resides in a single global virtual address space. Protection domains are implemented not by different address spaces, but instead using the segmentation and capabilities of guarded pointers [2]. In a SEND instruction, the destination (*DestAddr*) must be specified with an unforgeable virtual address pointer. Since the global translation look-aside buffer (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 that is certified to be safe, in that it never blocks indefinitely, and is guaranteed to not cause any unrecoverable errors. Certification may be done through careful human inspection or compiler analysis. In the M-Machine, the SEND instruction requires *HandlerIP*, which specifies the handler to be invoked, to 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. Therefore no authentication is required at the destination. To further enhance protection, 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.

5 Related Work

Mapping message interfaces into the processor's general registers was proposed by Henry and Joerg [7], who describe an interface where messages are buffered in a set of randomly-accessible registers. The MM interfaces are mapped to existing general-register names instead, thereby avoiding copying overhead and a separate register name space. As most message words are used only once by the handler, the MM extraction interface is optimized to automatically remove each consumed word from the message queue.

The multithreaded dispatch mechanism is an extension of the MIT J-Machine [8] model. As the J-Machine uses a streaming injection mechanism, the entire message injection phase must be treated as a critical region by convention to avoid message corruption. Mukherjee *et. al* [9] describe memory-mapped interfaces that exploit the processor's data cache for faster access to messages. We make a similar assumption when simulating the memory-mapped models, by emulating the memory-mapped network operations with cached memory operations and appropriate flush instructions.

To prevent message interception, the CM-5 [4] drains the network at the end of each time-slice, during which only threads from one *same* job are scheduled. Such *gang-scheduling* techniques have very high global synchronization overhead. The FUGU system [10] uses a hardware stamp to identify each message and thread. A message is presented to its receiver only if the correct receiver is currently installed. Otherwise, the operating system takes over. An analogous but insecure software-based tagging system is used in the revised Active Message interface [13] to match up messages with intended destination nodes, making it unlikely for an inadvertently misdelivered message to be accepted. To prevent starvation, FUGU allows the OS to revoke *interrupt-disables* and regain control if incoming messages are blocked for too long. The revised Active Message system instead relies on restrictions similar to those required of *trusted handlers* in the M-Machine, though no enforcement facility is provided. The FLASH system [11] provides virtual-memory mapped messaging facilities through the MAGIC co-processor, enforcing protection with the address translation layer. However, user-level handlers are unsupported on FLASH because the system is vulnerable to malicious handlers.

6 Conclusion

A fast, low-overhead message subsystem is essential for efficient fine-grain multicomputing. We compare alternative message interface designs using different dispatch and mapping mechanisms, and with either a buffered or streaming model. The dispatch system is found to be critical, as a dedicated-thread mechanism cuts dispatch latency by as much as $18\times$ when compared to an interrupt-driven interface. Mapping the message interface to memory, instead of integrating it with the processor (register/instruction-mapped), costs up to about $4\times$ higher end-to-end latency. For short messages, buffered and streaming models do not substantially differ, with as low as 10% difference. Although quite efficient, existing integrated interfaces do not meet the challenges or exploit the opportunities presented by emerging multithreaded processor architectures, which also require low-overhead mechanisms for protection from message corruption, interception and starvation.

We present the M-Machine message interface architecture, which features (a) atomic injection from a general-register mapped buffer, (b) dedicated message dispatch thread slot with no context-swap overhead, (c) streaming extraction through general register names. This configuration consistently shows lower processor occupancy (up to one-third less) than all other models. It is also often faster, achieving a round trip ping latency of 38 cycles, compared to 43 cycles for the buffered injection interface that is mapped to a dedicated register name space, and 103 cycles when the buffer is memory-mapped. Starvation-free multithreaded messaging is achieved through atomic injection from register-mapped buffers dedicated to each thread. Using unforgeable destination addresses and trusted message handlers, the system also features fine grain control over the messaging domain reachable by each thread, and the accessible remote operations.

With increasing demand for computing power, multicomputers 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. Growing 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 threads of control on each chip, such systems will encounter many protection, resource-allocation, and starvation issues, as demonstrated in the M-Machine. We have shown that the simple messaging mechanisms described in this paper can help provide a solution to these challenges.

Acknowledgment The research described in this paper 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. The authors wish to thank the anonymous reviewers for their valuable feedback.

References

[1] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, Whay S. Lee, "The M-Machine Multicomputer," in *Proceedings of the 28th Annual International Symposium on Microarchitecture, 1995.* pp. 104-114.

[2] Nicholas Carter, Stephen Keckler, William Dally, "Hardware Support for Fast Capability-based Addressing," in *ASPLOS VI.* 1994.

[3] James Laudon, Daniel Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server.", in *ISCA 1997.*

[4] Charles Leiserson *et. al.*, "The Network Architecture of the Connection Machine CM-5," in *SPAA 1992.* pp. 272-285.

[5] Michael J. Beckerle, "An Overview of the START (*T) Computer System", Technical Report MCRC-TR-28, Motorola CRC, Cambridge, MA. July 1992.

[6] Thorsten von Eicken, David Culler, Seth Copen Goldstein, Klaus Erik Schauser, "Active Messages: a Mechanism for Integerated Communication and Computation," in *ISCA 1992.* pp. 256–266.

[7] Dana Henry, Christopher Joerg, "A Tightly-Coupled Processor-Network Intrface," in *ASPLOS 1992.* pp. 111-122.

[8] William J. Dally *et. al.*, "The *J-Machine*: A Fine-Grain Concurrent Computer," in *IFIP 1989.* pp. 1147-1153.

[9] Shubhendu Mukherjee, Babak Falsafi, Mark Hill, David Wood, "Coherent Network Interfaces for Fine-Grain Communication," in *ISCA 1996*. pp. 247-258.

[10] Kenneth Mackenzie *et. al*, "Exploiting Two-Case Delivery for Fast Protected Messaging." in *HPCA 1998*. pp 231–242.

[11] Jeffrey Kuskin *et. al.*, "The Stanford FLASH Multicomputer," in *ISCA 1994*. pp. 302–313.

[12] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, "Protected User-Level DMA for the SHRIMP Network Interface", in *HPCA 1996*.

[13] Alan Mainwaring, David Culler, "Active Message Applications Programming Interface and Communication Subsystem Organization", U. C. Berkeley Technical Report CSD-96-918, October, 1996.