Message-Passing for the 21st Century: Integrating User-Level Networks with SMT

Mike Parker, Al Davis, Wilson Hsieh
School of Computing, University of Utah
{map@cs.utah.edu, ald@cs.utah.edu, wilson@cs.utah.edu}

Abstract

We describe a new architecture that improves message-passing performance, both for device I/O and for interprocessor communication. Our architecture integrates an SMT processor with a user-level network interface that can directly schedule threads on the processor. By allowing the network interface to directly initiate message handling code at user level, most of the OS-related overhead for handling interrupts and dispatching to user code is eliminated. By using an SMT processor, most of the latency of executing message handlers can be hidden. This paper presents measurements that show that the OS overheads for message-passing are significant, and briefly describes our architecture and the simulation environment that we are building to evaluate it.

Keywords: message-passing, I/O, SMT, user-level network interface, interrupt, general-purpose OS

Contact:

Mike Parker
50 S. Central Campus Drive
Room 3190 MEB
Salt Lake City, Utah 84112

Voice: (801) 587-9414
Fax: (801) 581-5843
E-mail: map@cs.utah.edu

Message-Passing for the 21st Century: Integrating User-Level Networks with SMT

Mike Parker, Al Davis, Wilson Hsieh
School of Computing, University of Utah

Abstract

We describe a new architecture that improves message-passing performance, both for device I/O and for interprocessor communication. Our architecture integrates an SMT processor with a user-level network interface that can directly schedule threads on the processor. By allowing the network interface to directly initiate message handling code at user level, most of the OS-related overhead for handling interrupts and dispatching to user code is eliminated. By using an SMT processor, most of the latency of executing message handlers can be hidden. This paper presents measurements that show that the OS overheads for message-passing are significant, and briefly describes our architecture and the simulation environment that we are building to evaluate it.

1 Introduction

The same VLSI technology forces that are driving processor interconnect are also having an impact on I/O architectures. As clock frequencies increase, high capacitance processor and I/O buses cannot keep pace. These buses, on and off chip, are being replaced by point-to-point links. I/O interfaces are starting to look much more like message-passing networks, as is evidenced by recent standards such as InfiniBand[25] and Motorola's RapidIO[27]. Communication over these point-to-point I/O networks can be viewed as low-level message-passing, where queries are sent to devices and responses are received from devices. Since technology trends force the hardware to use point-to-point links, there is an interesting opportunity to expose communication directly to user-level software through a message-passing interface. By looking at this opportunity from a systems point of view (from user-level software down to the hardware), we anticipate that we can dramatically reduce the costs for both processor-to-processor and processor-to-IO message-passing.

Our architecture for addressing this problem consists of the following combination of ideas:

- An SMT processor allows the overhead of message handlers to be effectively hidden.
- A network interface that supports user-level access can be tightly coupled to the CPU to avoid the overhead and latency of slower I/O buses. In addition, the network interface can directly dispatch user-level threads on the SMT processor, which eliminates OS involvement in the common case
- A message cache buffers incoming messages so that they can be accessed quickly by the processor, and acts as a staging area for outgoing messages.
- A zero-copy message protocol allows messages to be delivered directly to user-space without copying.

Not all of these ideas are new. For example, previous research has explored the use of user-level network interfaces[3,9,11,13,18]. However, this specific combination of features is unique, in that it exposes interrupts directly to user-level programs. The important aspect of our architecture lies in its support for user-level messaging (for both interprocessor communication and I/O) in a general-purpose operating system with small modifications to an SMT processor.

The combination of features in our architecture should reduce message handling overheads dramatically, without requiring gang-scheduling or forcing a change of the message notification model that is seen by the user-level software. The SMT processor, originally targeted to hide message latency,

makes it possible to overlap computation and communication without adding a secondary communication processor. The combination of a zero-copy protocol, a message cache, and user-level access to the network interface allows user code to communicate without the overhead of OS involvement or data copying. Finally, our integration of the network interface (NI) with the SMT allow the NI to communicate message arrival events back to the target thread without most of the overhead an interrupt-style notification would incur.

2 Message Notification Costs

Figure 1 shows how a message send and receive may look from a single node point of view on a machine that uses a kernel-mode network interface and traditional interrupts for message arrival notification. Sends, receives, and notifications all make passes through operating system code. Since the operating system code is unlikely to reside in the cache, these system calls result in cache misses.

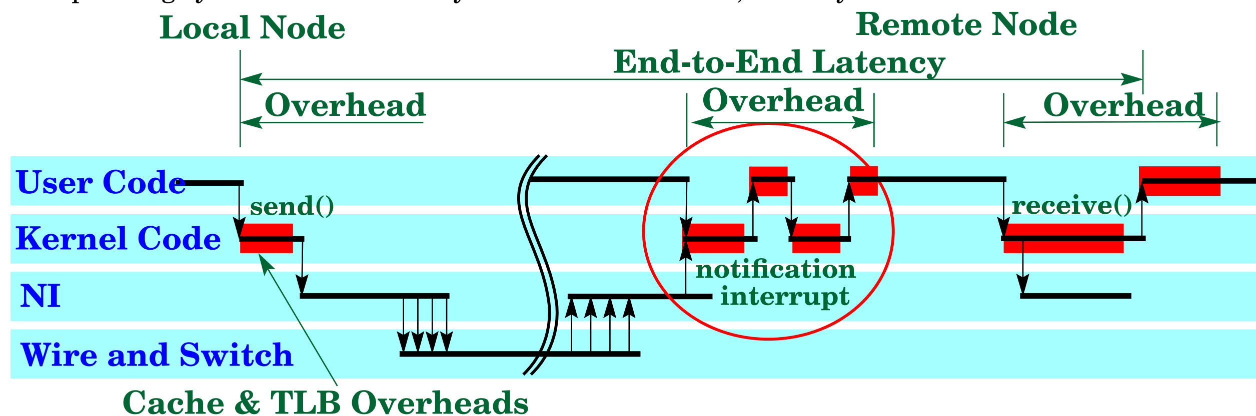

Figure 1: Anatomy of a message for a kernel-mode NI

User-level interfaces[3,9,11,13,18] and zero-copy protocols[5,7] significantly reduce the overhead of message sends and receives by eliminating operating system and copying overhead on the message send and receive sides. Notifications still have significant opportunity for optimization, as they remain the performance and scalability bottleneck in general multi-user environments. Polling for notifications consumes significant processor and memory resources, making them undesirable in a multiprogrammed system. Polling is especially poor for programs with irregular or unpredictable communication patterns. Interrupts in current architectures and operating systems are costly in terms of the number of processor cycles consumed to determine the cause of and handle the exception[12]. This makes them less than optimal for message notifications.

The components of an interrupt-style notification overhead include:

- processor pipeline flushing (due to the interrupt)
- serial instructions to get and save processor state
- cache and TLB misses to bring in OS code and data to determine the cause of the interrupt
- reading NI registers or data structures to determine which process should be notified
- posting the notification to the process via a signal or other such mechanism
- cache, TLB and context switch overhead to begin execution of the user-level notification (signal) handler
- a trap to return from the user-level handler back to the OS
- serial instructions to save processor state
- cache and TLB overhead to bring in OS code and data
- scheduler and context switch overhead to bring back in the original user process
- post kernel cache and TLB overhead to bring back in the user process's instructions and data

Using a refined version of Schaelicke's interrupt measurement work[22], we measured the overhead of servicing a network interrupt for a minimum sized packet. Under Solaris 2.5.1 on a 147-MHz Ultra 1, such an interrupt takes approximate 119 microseconds (17500 cycles) when user-level code is utilizing the entire L2 cache. The process of handling such an interrupt results in about 380 kernel-induced L2-cache misses. (Fewer misses may be observed in practice if the user-level code is not utilizing the entire L2 cache.) Assuming that each cache miss takes an average of approximately 270 ns to service[17], this accounts for about 103 microseconds or 87% of the interrupt processing time. The remaining 13% of the time is spent in flushing the pipeline after the interrupt and trap, carefully reading and saving critical processor state, querying the NI for information about the interrupt, and executing operating system code to determine how to deal with the interrupt. In addition to incurring the overhead of cache misses during an interrupt, the process that was running when the interrupt occurred could see up to another 380 L2 cache misses once it is re-scheduled after the interrupt to refill the cache with its working set.

L2-cache and TLB miss penalties unfortunately scale at memory speeds, as opposed to processor speeds. As a result, these overheads will become even more important as the memory gap widens. Optimizations to the OS and signalling system can reduce this overhead. However, reducing the number of cache misses and other overheads to get the OS penalty down below a few microseconds does not seem plausible. To make frequent notifications acceptable, the operating system's involvement must be significantly reduced or eliminated.

3 Architecture

Notifications only become the bottleneck when the rest of the message-passing system is appropriately tuned. This section describes the system architecture, showing how it is optimized for efficient messaging, describes how notifications are delivered to the user-level process without kernel involvement, and walks through the path a one-way message takes though this architecture. Figure 2 shows a block-level view of the architecture.

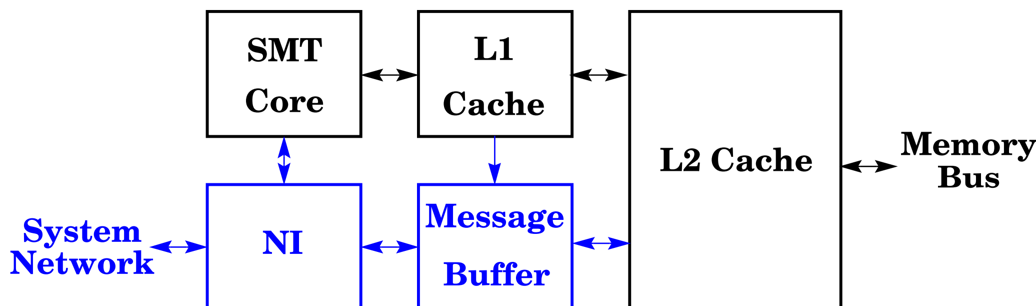

Figure 2: Block-level diagram of our architecture

3.1 Components of the Architecture

Using an SMT processor allows communication-related threads to run parallel to computation-based threads. Much of the overhead required to process sends, receives, and message arrival notification can be hidden by overlapping these functions with computation. Previous work has dealt with overhead by providing external communication processors[14,21]. These extra processors add to the overall latency and complicate the message-passing mechanism due to the additional overhead of communication between the computation and communication processor. The simultaneous nature of the SMT processor makes it possible to provide this overlap without requiring the use of an extra communications processor.

The message buffer acts as both a staging area for outgoing messages as well as a cache for incoming messages[23]. Messages may be composed directly in the message buffer for user-program written (PIO-style) transfers, or fetched from user memory for DMA-style transfers. Buffering outgo-

ing messages in the message buffer allows send data to be prefetched from memory and buffered before going out on the network. This buffering reduces the probability that the network will need to be stalled, tying up network resources, while waiting for outgoing message data to be supplied by the local memory subsystem.

Incoming messages are placed in the message buffer. The message buffer acts as a cache for incoming message data. As a message arrives, the message buffer invalidates corresponding cache lines in the L1 and L2 caches. Misses in the L1 cache result in concurrent lookups in both the L2 cache and the message buffer. In this way, the message buffer is similar to a victim cache to the L2 cache. When there is a cache hit in the message buffer, data is supplied directly to the L2 cache. This cache has a triple effect. First, it reduces overhead at the memory interface by saving the data two trips across the memory bus. Second, it keeps the data near the CPU, where it can be provided quickly on demand, thus reducing the overall end-to-end latency. Third, having a separate message cache avoids polluting the cache hierarchy because the processor's working set is not evicted by incoming messages.

A user-level accessible NI is used to reduce send and receive overhead. Having the network interface on the same die, possible in the System on a Chip (SoC) era, opens up possibilities to more tightly integrate it with the processor core, further reducing overhead and latency. Having the NI on die gives the processor access to it on a per cycle basis. This close coupling further reduces the overhead in getting information to and from the NI. Message sends and receives do not have to go out over slow and inefficient I/O buses. A zero-copy protocol[5,7] is used to eliminate copying overhead for received messages. The combination of user-level access to a closely coupled NI and the zero-copy protocol allow for efficient sends and receives.

3.2 User-level Notifications

Part of the inefficiency of interrupt processing is due to the legacy view that interrupts are expected to be infrequent. In a fine-grained message-passing environment that uses interrupts for notifications, this is not the case. One of the contributions of this work is to provide a mechanism whereby the network interface can directly deliver notifications to a user-level process without the aid of the operating system. This is accomplished by allowing the NI to share some control over thread execution.

Having the NI tightly coupled to the CPU makes it possible for the NI to share some control over process execution in much the same way as load-store units can control the pipeline as cache misses are detected. This can be done by thinking of interrupts and notifications differently. Notifications are a way of telling the user process that it needs to either wake up (in the case of polling) or stop what it is doing (in the case of an interrupt) and deal with a message arrival. One analogy to this would be cache misses in a modern superscalar processor. When a cache miss occurs, the memory reference instruction stalls. Other independent instructions may continue, but the instruction that caused the miss waits for the cache to fetch the relevant data. Once the data has been fetched, the cache returns it to the processor core, and the processor again places priority on the memory reference and dependent instructions.

This basic idea can be extended to message arrival notifications. When a process reaches a point where it needs a message arrival notification, it can tell the hardware what specific process state to change (i.e., program counter or runability) when an arrival occurs. It can then continue processing until either the notification occurs, or it runs out of things to process. When the notification finally takes place, the hardware can redirect the user-level software to work on processing the message arrival.

To give the NI shared control over user processes, three mechanisms are available in our architecture.

- If a thread wishes to be notified when a message arrives, it can set a lock in a hardware synchronization lock table[24]. The NI can clear the lock bit upon message arrival, which releases the process to run again. If the associated thread is not currently in the CPU, the OS

receives the notification, and sets the appropriate hardware state to notify the thread the next time it is scheduled.

- Just as an interrupt causes a current processor to asynchronously branch into a kernel level interrupt handler, the NI can cause a running user-level process to asynchronously branch to a notification handler.
- Upon message arrival, the NI can schedule a new thread on the SMT with a previously setup context. This new thread starts in either an unused context on the SMT or it can evict a running thread, according to OS policy. If no contexts are available, the NI would notify the OS, so that it could create the context and schedule it to run at a later time.

3.3 The Journey of a Message

Figure 3 shows how a message may look in our architecture.

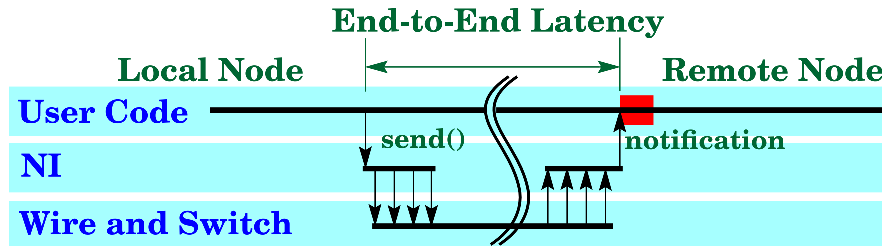

Figure 3: Anatomy of a message in our architecture

A user-level program that wants to send a message to a process on a remote processor first composes the message to be sent. Message control information is written into the message buffer. It includes information on how the message is to be handled on the remote end (where it will be placed and whether to notify the receiving process), a small amount of user-defined meta-data, and an optional local pointer to message data. The user process then directs the NI to send the message. The NI begins to prefetch message data from local memory into the message buffer if required, fills in information such as routing information, and begins streaming the message onto the wire. On the receive side, the message is placed directly in the address space of the receiving process by the NI. While the message is arriving, the NI places the message into the message buffer. As it fills cache lines in the message buffer, it acquires ownership of those cache lines though the L2-cache interface and sets up appropriate cache tags in the message buffer.

When an entire message arrives, the NI fills in a notification structure in the user process's memory space. It then determines which method should be used to notify the corresponding process of the message arrival. In the case of a blocked process waiting on a message arrival, the NI clears a lock bit in a synchronization table, which makes the corresponding thread runnable. The user-level process begins executing and handles the incoming message. In the case of an asynchronous branch or user-level interrupt, the SMT switches to an alternate program counter and stack. User-level code is then responsible for saving any of its own state as necessary before handling the notification. Finally, in the case of a created thread, the NI gives the SMT core minimal context, including a program counter, stack pointer, and a pointer to the notification structure. The SMT core begins executing at the given program counter.

In summary, we have presented an architecture that significantly reduces send, receive, and notification overhead. We have presented three separate user-level notification mechanisms, and have walked through the path a message takes in our architecture. The key features of the architecture are the following: SMT processors hide and tolerate message overhead and latency, send and receive overhead is reduced by a user-level network interface combined with efficient protocols, and notifications can be delivered directly to user-level without the overhead of an operating system.

4 Related work

Simultaneous Multithreading (SMT) architectures[8] have begun to see commercial attention.The SMT processor is targeted to tolerate latency and hide overhead by allowing one thread to process overhead or wait for a long latency operation while the execution of independent threads continues. SMT architectures promise the ability to simultaneously take advantage of both ILP and thread-level parallelism within a single processor core. This architecture helps pave the way to more efficient communication and synchronization of threads.

Dean Tullsen et al.[24] shows how extra lock and release hardware can be introduced to provide fine-grained synchronization for threads within the SMT processor. This efficient locking mechanism allows one thread to block on a hardware semaphore and be released by another co-operating thread quite efficiently. One of the suggested notification primitives in this paper extends some of the control over this hardware locking table to external events, such as message arrival notifications from the NI.

Several previous systems have advocated moving the NI closer to the CPU. Flash[14], Avalanche[23], Alewife[1], Shrimp[3], and Tempest[21] all placed the NI directly on the system memory bus. Moving the NI to the system bus significantly reduces the cost of accessing the NI over accessing it on a less efficient I/O bus. In addition to reducing overhead, placing the NI on the system bus allows these systems efficient access to coherency traffic, which several of these systems use to an additional advantage. The MIT J-Machine[6] and M-Machine[11] take it one step closer by bringing the NI directly onto the custom processor. Alewife, the J-Machine, and the M-Machine also have an interesting characteristic in common in that they all use a thread model or a thread-like model to deal with communication. Alewife uses a modified SPARC processor in an unconventional way to implement these threads.The J-Machine and M-Machine both build a custom processor to get the desired thread behavior. SMT processors now seem to be a natural way to achieve effective functionality of these machine with only minor modifications to CPU structure.

In many ways the architecture in this paper is similar to the M-Machine. Both take advantage of thread capable processors to hide message overhead, and both have forms of automatically dispatching threads when a message arrives. Our architecture differs from the M-machine in the following ways. Messages are received directly into a users address space via hardware, eliminating the need for trusted message handlers. Incoming messages are placed into a message buffer, or message cache, to avoid pollution of the processor's cache hierarchy. As a part of our work, we are evaluating the usefulness of this message-passing architecture both in the context of parallel processing and in the context of network-based IO. Finally this architecture is built upon modifications of upcoming SMT architectures.

Avalanche[23] placed the network interface on the system bus, keeping it close to the processor. This allowed it to participate in coherency traffic, and thus maintain a local network cache. The local cache enables the Avalanche network interface to supply network data to the processor more quickly than main memory. In addition, it avoids the overhead of wasting system bus bandwidth to transfer message data across the system bus twice; once on the way to main memory on message arrival, and once on the way back to the processor when the message is consumed.

For Hamlyn[5] Wilkes proposed sender-based protocols to reduce overhead. Having the sender manage its destination buffers implies that data can be easily and effectively received directly into the receiver's process space. Avalanche also used a sender-based messaging protocol (DDP)[7] to reduce overhead. Sender-based protocols allow simple and efficient hardware to place incoming messages directly into the receive process's address space. This avoids kernel involvement on receives. Unlike Hamlyn and DDP, the sender-based portion of the protocol in this work uses virtual addresses in combination with an NI TLB to remove restrictions on receive buffers.

Active Messages[10] embed a message handler in the header of a message. When a message arrives, the message handler is executed to handle the payload of the message. Though it is not specifically a goal of this work, the architecture described here would support Active Messages rather well.

A thread waiting for a message could immediately jump to the handler code in the header without the penalty of an interrupt and without interfering with the currently running thread. If messages are received directly into a message cache, then this handler code could potentially execute directly out of the message cache, also saving the cache overhead of bringing in conventional handler code. Illinois Fast Messages[20] is effectively a platform independent implementation of Active Messages.

U-Net[9] reduces communication overhead and latency by virtualizing the network interface. The local process communicates with the network interface by placing and picking up message packets from per-process send and receive queues. U-Net suggests placing a TLB in the network interface to avoid the added restriction of fixed pinned pages. The architecture in this paper also gives the NI access to a TLB to allow sends and receives to be handled in user-space.

5 Conclusion

SMT allows important computation to continue while interprocessor communication and I/O processing and communication overhead is handled in the background. Since message latency is similar to memory latency, one way of viewing this work is using an architectural technique for hiding memory latency to hide message latency. Conversely, we can view our work as generating more parallelism for SMT processors from I/O and parallel workloads.

To evaluate this architecture, we are extending LRSIM[22] to accurately model an SMT processor and adding a model of our network interface. LRSIM is based on RSIM[19], and has already been extended to include accurate I-cache, memory and I/O architecture models. The simulator includes a fairly complete NetBSD based kernel that will be extended to handle the SMT processor (all kernel operations are fully simulated). The simulator runs unmodified Solaris binaries. For our evaluations, we will model a 2-5 GHz 4-8 thread SMT that can issue 8-16 instructions per cycle. L1 instruction and data caches will be 32KB to 128 KB each, and the L2 Cache will be 4-16MB. The system network bandwidth modeled will range from 4Gb/s to 32Gb/s. Disk controllers, LAN interfaces (i.e., Ethernet), and other I/O devices will hang off the system network. There are a few open issues in our design:

- It remains unknown how much send side buffering will be optimal. Too little buffering could lead to too many bubbles in the network fabric. Too much could lead to increased message latency. For the purposes of this design, the amount of buffering will be user configurable.
- The point at which DMA becomes more efficient than PIO needs to be characterized for this architecture. Though it has been characterized for previous systems, the break-even point on our architecture may be different as PIO can overlap computation on the SMT processor.
- The relative performance of each of the notification mechanisms needs to be characterized.

The results from the simulations will be compared with existing and proposed systems to assess the benefits of our architecture.

Acknowledgements

We thank Lambert Schaelicke for his help in understanding and extending his interrupt measurement techniques. We also thank Lambert, John Carter, Katie Parker, and Greg Parker for help in reviewing this paper. This work was sponsored in part by DARPA and AFRL under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the U.S. Government.

References

[1] Anant Agarwal, et al. The MIT Alewife Machine: Architecture and Performance. In Proceedings of the 22nd Annual ISCA, 1995, pp. 2-13.

[2] Mark Birnbaum and Howard Sachs. How VSIA Answers the SoC Dilemma. IEEE Computer, 32(6):42-50.

[3] Matthias A. Blumrich, et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In Proceedings of the 21st Annual ISCA, April 1994, pp. 142-153.

[4] Doug Burger. Billion-Transistor Architectures. IEEE Computer, 30(9):46-48, September 1997.

[5] Greg Buzzard, et al. Hamlyn: a high-performance network interface with sender-based memory management. HP Laboratories Technical Report HPL-95-86, August 1995.

[6] William J. Dally, et al. Retrospective: The J-Machine. In 25 Years of ISCA - Selected Papers, 1998, pp. 54-58.

[7] Al Davis, Mark Swanson, and Mike Parker. Efficient Communication Mechanisms for Cluster Based Parallel Computing. Communication, Architecture, and Applications for Network-Based Parallel Computing, 1997, pp. 1-15

[8] Susan J. Eggers, et al. Simultaneous Multithreading: A Platform for Next-Generation Processors. IEEE Micro, 17(5):12-19, October 1997.

[9] Thorsten von Eicken, et al. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In Proceedings of the 15th ACM SOSP, December 1995, pp. 40-53.

[10] Thorsten von Eicken, et al. Active Messages: A Mechanism for Integrated Communication and Computation. In Proceedings of the 19th Annual ISCA, 1992, pp. 256-266

[11] Marco Fillo, et al. The M-Machine Multicomputer. In Proceedings of the 28th Annual International Symposium on Microarchitecture, 1995, pp. 146-156.

[12] Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. In Proceedings of 1999 USENIX Annual Technical Conference, FREENIX Track, June 1999.

[13] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In Proceedings of the 5th International ASPLOS, October 1992, pp. 111-122.

[14] Jeffrey Kuskin, et al. The Stanford FLASH Multiprocessor. In Proceedings of the 21st Annual ISCA, April 1994, pp. 302-313.

[15] Richard P. Martin, et al. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In Proceedings of the 24th Annual ISCA, June 1997, pp. 85-97.

[16] Doug Matzke. Will Physical Scalability Sabotage Performance Gains? IEEE Computer, 30(9):37-39, September 1997.

[17] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In Proceedings of the USENIX 1996 Technical Conference, January 1996, pp. 279-294.

[18] Shubhendu S. Mukherjee and Mark D. Hill. Making Network Interfaces Less Peripheral. IEEE Computer, 31(10), October 1998, pp 70-76.

[19] V. S. Pai et al. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In Proceedings of the 3rd Workshop on Computer Architecture Education, 1997.

[20] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. IEEE Concurrency, vol 5, no. 2, April-June 1997, pp. 60-73.

[21] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In Proceedings of the 21st Annual ISCA, 1994, pp. 325-336.

[22] Lambert Schaelicke. Architectural Support for User-Level I/O. Ph.D. Dissertation, University of Utah, 2001.

[23] Mark Swanson, et al. Message Passing Support in the Avalanche Widget. Technical Report UUCS-96-002, University of Utah, March 1996.

[24] Dean M. Tullsen, et al. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In Proceedings of the 5th HPCA, January 1999.

[25] InfiniBand Trade Association. http://www.infinibandta.org.

[26] International Technology Roadmap for Semiconductors. Semiconductor Industry Assoc., 1998.

[27] Motorola Semiconductor Product Sector. RapidIO: An Embedded System Component Network Architecture. February 22, 2000.