Reducing Waiting Costs in User-Level Communication

Stefanos N. Damianakis, Yuqun Chen, Edward W. Felten
Department of Computer Science
Princeton University
Princeton, NJ 08544 USA
{*snd,yuqun,felten*}*@cs.princeton.edu*

Abstract

This paper describes a mechanism for reducing the cost of waiting for messages in architectures that allow user-level communication libraries. We reduce waiting costs in two ways: by reducing the cost of servicing interrupts, and by carefully controlling when the system uses interrupts and when it uses polling. We have implemented our mechanism on the SHRIMP *multicomputer and integrated it with our user-level sockets library. Experiments show that a hybrid spin-then-block strategy offers good performance in a wide variety of situations, and that speeding up the interrupt path significantly improves performance.*

1. Introduction

Many network interfaces can place incoming data directly in user memory [1, 7, 3, 2]. This capability enables the construction of very efficient network software since the network interface can deliver a burst of packets without any software intervention. On such an architecture, communication can be handled entirely in a user-level library.

In message-passing systems, software overhead is attributed to the *send* and *receive* primitives. Efficiently dealing with send is straightforward because the sending process is always running and able to send data. The key performance issues for send are avoiding message copies and kernel system calls.

The receive mechanism is the difficult operation in user-level communication architectures. When a message arrives, the receiving process may not be running or, worse still, execution may not have reached the appropriate receive call, so the receiver may not have specified a destination address (to avoid message copying). When a process does call receive, if a suitable message has not yet arrived there is no way to tell when one will arrive.

To find out that new data has arrived the user-level library must either poll, or block and be awakened later by an interrupt. This requires both a policy for when to poll and when to block, and a mechanism for efficient blocking. This paper considers the questions of which receive policy and which mechanism to use.

We present an implementation on the prototype *SHRIMP* multicomputer [1, 7], and the results of experiments using our user-level sockets library [5], for micro-benchmarks, larger benchmarks, and for a distributed file system. Our results show that a hybrid spin/block policy is best in a wide range of situations, and that reducing the interrupt service overhead significantly improves performance.

2. Architecture

Our approach makes the following two assumptions about the communication architecture:

- the network interface (NI) delivers incoming data directly to user memory. No explicit receive is required by user or kernel code.
- the receiving node can dynamically enable and disable receive interrupts at page granularity[1]. In other words, incoming messages may or may not interrupt the receiver, and the receiver has fine-grain control over when interrupts occur.

These assumptions imply that user-level communication software on the receiver can be given responsibility for detecting the arrival of new data. It can achieve this by polling or, with appropriate kernel support, by blocking the process and activating interrupts for specific incoming data.

[1] This requirement is not strict; some of our implementations use interrupts and some do not.

Some existing network interfaces that directly support these assumptions are *SHRIMP*, Hamlyn[3], and Telegraphos[12]. In addition, other architectures, such as Myrinet[2] and FLASH[10], are sufficiently programmable to satisfy the assumptions.

2.1. The *SHRIMP* Architecture

The *SHRIMP* multicomputer[1, 7] consists largely of off-the-shelf components. The nodes are Pentium PCs running the Linux operating system, which communicate through a routing network via a custom designed network interface card [1]. The *SHRIMP* network interface closely cooperates with a thin layer of software to form a communication mechanism called virtual memory-mapped communication (VMMC) [6]. This mechanism supports various message-passing packages and applications effectively, and delivers excellent performance [7].

The prototype system consists of four interconnected nodes. Each node is a DEC 560ST PC with a 60 MHz Pentium CPU. The routing network is an Intel backplane consisting of a two-dimensional mesh of Intel Mesh Routing Chips (iMRCs), and is the same network used in the Paragon multicomputer. The backplane supports deadlock-free, oblivious wormhole routing, and preserves the order of messages from each sender to each receiver.

The receive-side network interface transfers data from network packets directly to user memory. In addition, if the sender has requested (and the receiver agrees), the network interface interrupts the processor either after the data has been deposited. Further, the receiver can control the interrupts by enabling or disabling them for any page that has been mapped to receive incoming data.

2.2. *SHRIMP* Communication Software

We have built several communication libraries for *SHRIMP*, implementing various standard communication interfaces. All of these libraries have a common structure [7] based on pairwise connections between processes. A connection generally consists of some buffers for transmitting the contents of user messages, and some buffers for transmitting control information. To pass a message, the sender first chooses a remote data buffer into which to place the message contents. After transferring the message contents, the sender transmits some control information to tell the receiver that the data is in place, and possibly to provide descriptive information such as the message size. Since *SHRIMP* guarantees in-order delivery of packets, the control information arrives at the receiver after the data.

To receive a message, a process looks for incoming control information that denotes the arrival of a suitable message. Once this is found, the receiver can consume the message data. The receiver then sends some control information back to the sender, to signal that data buffer space can be re-used.

The task of looking for incoming control information motivated the work described in this paper. Our experience in building communication libraries [7] for *SHRIMP* shows that changes in control data (*flags*) indicate the arrival of new data. As a consequence, our implementation provides support for waiting (via spinning, blocking or a combination) for any of a set of flag values to be changed.

3. Strategy

Our design tries to improve performance in two ways: by reducing the cost of interrupt service, and by taking interrupts only when it is helpful to do so.

3.1. Reducing Interrupt Service Time

To make interrupt service faster, we have to reduce the amount of work done as a result of the interrupt. Although many systems execute a user-level handler when messages arrive, doing this adds overhead (at least $10\mu s$ on our hardware) that isn't necessary for building user-level communication libraries. In early implementations of our system, we found ourselves installing null handlers and using notifications (user-level interrupts) merely to awaken sleeping processes.

All that is really needed is a way to reawaken a blocked process when a suitable message arrives. If the user process blocks waiting for message arrival, then it will certainly look for the message immediately upon reawakening. There is no reason to disrupt the flow of control by executing a handler.

Our new mechanism adds a system call that blocks the calling process until specific data arrives. The user-level process specifies which memory locations it is watching, and the device driver interrupt handler wakes up the process when one of those locations is modified by an incoming message.

3.2. Controlling the Number of Interrupts

Our second goal is to carefully control when interrupts are taken. Our ultimate goal is to minimize the amount of CPU time (including time spent in the kernel for context switching and interrupt service) that a process uses to detect the arrival of messages. This time is the sum of the time the process spends spinning while waiting for messages to arrive, and the time it spends in the kernel for context switches, system calls, and interrupt processing when the process decides to block.

This tradeoff is much like the one between spinning and blocking in the context of multiprocessor locks. Ousterhout [13] describes spin-blocking for locks in the Medusa system. Karlin et al. [9] study the performance impact for several algorithms. The general discussion in this section is based on Karlin's work, though many of the implementation tradeoffs in Karlin's case are different than in ours.

The right choice between spinning and blocking depends on the *arrival delay*: the length of time before the next message will arrive. The cost of spinning is equal to the arrival delay, while the cost of blocking is a constant T_{block}, independent of arrival delay, equal to the sum of two context switch times, plus the time to service a message-arrival interrupt, plus the time to change the hardware state to enable and disable interrupts.

If the process could predict what the arrival delay was going to be in each case, it could follow the *optimal off-line* policy: spin if the arrival delay will be less than T_{block}, and block if the arrival delay will be more than T_{block}. In reality, of course, there is no way to predict the arrival delay, so the optimal off-line policy is not realizable, though it does provide a good baseline for evaluating practical policies.

The two simplest policies are `spin` and `block`. Each is much worse than optimal in some cases. If the arrival delay is very long, `spin` uses a lot of CPU time spinning when blocking would have been much better. If the arrival delay is very short, `block` incurs a significant constant overhead when spinning would have been much better.

A simple hybrid `spin-block` policy works fairly well: spin for time T_{block} and then block if the message has not yet arrived. This policy does well in the two limiting cases of very small and very large arrival delay. Karlin et al. show [9] by a simple argument that this policy is 2-competitive: its cost is never more than twice that of the offline optimal policy, regardless of the sequence of arrival delays.

3.3. Strategies

We identify seven strategies that processes can use when waiting for messages:

- *spin*: sit in a loop polling for message arrival;
- *sched*: sit in a polling loop, but yield the processor after each iteration of the loop;
- *block*: block the process and use interrupts to reawaken it;
- *notify*: block the process and use a user-level handler to reawaken it;
- *spin-sched*: spin for a while and then switch to the sched policy;
- *spin-block*: spin for a while and then switch to the block policy;
- *spin-notify*: spin for a while and then switch to the notify policy.

Strategy	User Spin	Interrupts	System Calls	Kernel Reschedules per System Call
Spin	✔		none	–
Block		✔	always	1
Spin-Block	✔	✔	sometimes	1
Notify		✔	always	1
Spin-Notify	✔	✔	sometimes	1
Sched			always	≥ 6
Spin-Sched	✔		sometimes	≥ 3

Figure 1. *Summary of the differences between the various strategies*

In addition to studying the tradeoffs between spinning and blocking, these strategies enable us to analyze the performance of three different mechanisms that detect data arrival: fast-interrupts, user-level notifications, and the kernel scheduler. The table in Figure 1 illustrates the main differences between the various strategies.

4. Implementation

This section describes how the seven strategies are implemented. Some of the strategies are easy to implement; for example *spin* is a simple polling loop that waits for data to arrive. We focus here on the *spin-block* strategy. Implementations of the other strategies are either straightforward, or closely related to the implementation of *spin-block*.

Spin-block is implemented using the following three components:

- **user-level function**: called by the user-level communication library; it spins as required by the spin/blocking strategy, and then makes a system call to block.

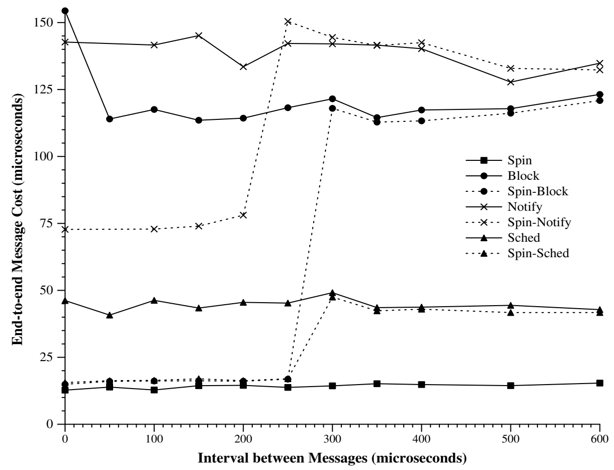

Figure 2. *Performance of a single ping-pong program for each strategy with message delay ranging from* 0 *to* $600\mu s$.

- **kernel system call**: if data has not arrived, this system call enables interrupts for the specified pages and then blocks the process until a suitable message has arrived.
- **device driver interrupt handler**: we added a hook to the standard *SHRIMP* interrupt handler to check whether data has arrived for a page that a process is waiting on. If so, the interrupt handler clears the interrupt enable bit for all of the pages the process is waiting on, and wakes up the process. Otherwise, it proceeds to service the interrupt normally.

Together, the three pieces implement an efficient mechanism by which a user-level process may wait for one of a set of flags to be changed by the arrival of a message. Our technical report [4] contains a more detailed explaination for each piece, as well as a justification of the correctness of this code.

5. Experiments

This section describes the performance measurements we obtained using our mechanism. All tests were performed on a prototype four-node *SHRIMP* multicomputer.

5.1. Micro-Benchmarks

Our first set of experiments uses a simple client/server program which ping-pongs one word between two processes, running on two different *SHRIMP* nodes. The ping-pong program was written using the Shrimp Base Library (SBL) API, the lowest level communication API available on *SHRIMP* [6]. In these experiments we inserted a fixed delay (the "message delay") before the send operation in the ping-pong program. This simulates a case where some amount of computation is required before replying to each arriving message. By varying this delay time, we can see how hybrid strategies such as `spin-block` perform compared to the pure strategies. The graph in Figure 2 shows the performance for all of the spinning strategies when the message delay varies from 0 to $600\mu s$.

As expected, the pure `spin` strategy has the lowest cost per message, because each ping-pong process is the only runnable process on its host system and therefore is always ready to catch the incoming message. As a variation of pure spinning, `sched` performs not much worse, with some overhead due to making the system calls and calling the kernel scheduler. The two pure blocking strategies, `notify` and `block` show higher waiting time per message due to the cost of fielding the interrupts; a specialized system call in the case of `block`, and invocation of a user-level signal handler in the case of `notify`.

The hybrid strategies, `spin-sched`, `spin-block`, and `spin-notify` each spins for a fixed spin delay of $300\mu s$ before switching to the respective pure strategy.[2] As a result, they perform very well when the message delay is smaller than the spin delay. `spin-notify` has a constant overhead for handling user-level notifications even though they are not used during the spin-waiting phase. When the message delay gets larger than the spin delay, the hybrid strategies cannot catch the pingpong messages within their spinning phase and have to call on their respective pure strategy for receiving the message.

This experiment does not show the advantages of `sched`, `block` and `notify`. These policies give up the CPU, allowing another process to make progress. Sharing the CPU is especially important when several processes on a node are actively communicating.

Effects of multiprogramming We next ran five pairs of ping-pong processes simultaneously using a random message delay randomly distributed between $0 - 300\mu s$. The results shown in Figure 3 represent the one process average over at least four runs of each experiment. The left bars indicate the average per-process wall-clock time, and how much of it was user and system time. The right bar measures the average number of times the kernel scheduler was invoked.

[2]This number is our approximation of T_{block} described in Section 3.2

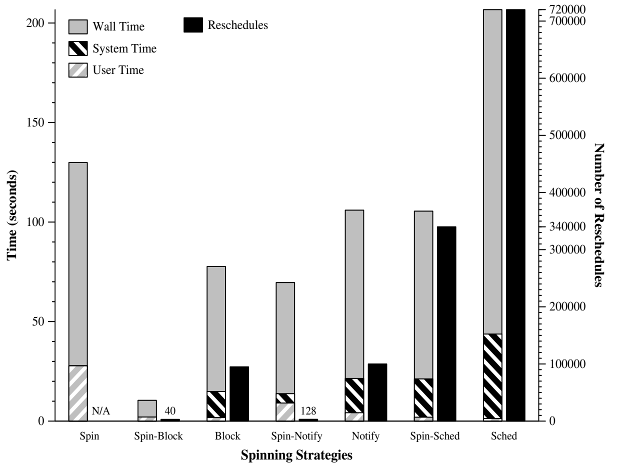

Figure 3. *Performance for 5 pairs of processes ping-ponging 100,000 messages with a random delay of* $0 - 300\ \mu s$ *between messages. The spinning strategies use a spin delay of* $300 \mu s$.

Our starting point is the pure `spin` strategy. Each process spins at user level waiting for data to arrive. Unfortunately, while one process spins, the other processes cannot receive messages or perform computation.

Clearly a process should give up the CPU when it is not using it. This leads to the three basic strategies: `block`, `notify`, and `sched`. In Figure 3 we see that `block` does better than `notify` despite an equal number of sleeps. The 36% increase in running time is due to the additional overhead required for notifications, i.e., the additional time required to invoke the user-level handler versus simply waking up a process. `sched` is still worse because the kernel scheduler thrashes; without interrupts, there is no other way to detect the arrival of new data.

Each of these basic strategies can be improved by adding a user-level spin (i.e. a check for new data) before yielding the CPU. Roughly speaking, the hybrid strategies perform like `spin` when the message delay is less than the spin time, and like `sched`, `block`, or `notify` (for `spin-sched`, `spin-block`, and `spin-notify`, respectively) when the message delay is larger. In this way, each hybrid approach improves performance. `spin-sched` reduces `sched`'s CPU yields (and, correspondingly, the running time) by 50%. `spin-block` and `spin-notify` virtually eliminate all CPU yields. Their difference in running time is because `spin-notify` must still service one notification with each message while `spin-block`, in the common case, runs without interrupts enabled.

`spin-block` performs particularly well in this experiment because it tends to cause processes that communicate with each other to run at the same time. When communicating processes are co-scheduled, they can exchange many messages without rescheduling. The co-scheduled processes receive messages while they are still spinning. If a quantum expires and one of the processes is descheduled, the other process quickly blocks and allows the system to find its way back into a co-scheduled state.

The kernel's scheduling policy has a significant impact on the co-scheduling phenomenon. We are currently studying the complex interaction between these two components.

`ttcp` is a public-domain benchmark that measures network performance using a one-way communication test in which the sender continuously pumps data to the receiver. We ran `ttcp` on top of our user-level communication library implementing the standard sockets API [5]. `ttcp`'s results are similar to the micro-benchmarks'; they are omitted because of space limitations, but are presented in out technical report [4].

5.2. Distributed File System

Our next set of experiments uses a distributed file system [14] implemented on top of our user-level sockets library. The distributed file system runs one process on each node; each process runs several user-level threads internally, and the processes communicate via our user-level sockets library. The distributed file system makes heavy use of `select` to wait on several sockets at the same time; this translates, in our sockets library, into waiting for one of several flags to change.

We ran a simple file system benchmark in which the file system process on one node accesses a sequence of file blocks that are initially cached in the memory of the file system process on the other node. This causes a sequence of data transfer messages and cache consistency protocol messages between the two nodes.

We measured the distributed file system performance only for the `spin`, `block`, and `spin-block` policies.

We first ran the file system benchmark alone, for each policy. We next ran the file system benchmark concurrently with a "CPU hog" process running in the background on each node. The `hog` processes simply spins in a tight loop consuming CPU cycles. We ignored the performance of the `hog` processes and measured the performance of the file system processes. Specifically, we measured the user and system time used by the file system processes, both with and

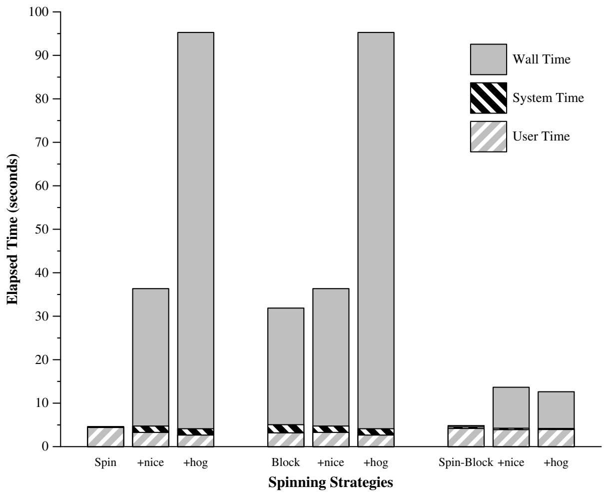

Figure 4. *Performance of a distributed file system benchmark.*

without hog processes. The difference between these times is the extra cost incurred by the file system processes because of interference from these CPU hogging processes. Next, we raised the CPU-scheduling priority of the file system processes so that they would be run in preference to the background processes. We then measured the wall-clock time and user and system time for the file system processes, both with and without the `nice` background processes. This experiment allows us to determine how much time is lost by communicating high-priority processes because of the existence of low-priority processes.

The results are shown in Figure 4. The `spin-block` strategy achieves the best performance in all three test cases, while the other two perform badly when either the `nice` or `hog` process is running. In the first case where the file system process is the only running process on a node, `spin` is obviously the optimal strategy (see Figure 2). Blocking in this case just adds unnecessary overhead. Therefore both `spin` and `spin-block` outperform the `block` strategy.

When `nice` or `hog` are also run, the file system process has to wait for the background process to complete a full scheduling quantum before being able to do any useful work. In particular, if the file system process is descheduled before a message arrives, it has to wait a full scheduling quantum before it can receive the data. Meanwhile, the file system process on the other node either exhausts its scheduling quantum, or, decides to use the `block` strategy and goes to sleep. This leads to a vicious cycle of scheduler thrashing and it explains why both `spin` and `block` strategies perform badly in the presence of a background process.

In contrast, the `spin-block` strategy has the tendency to co-schedule communicating processes. This property allows `spin-block` to avoid, or quickly escape thrashing cycle. This results in excellent performance for `spin-block` with both `hog` and `nice` processes.

6. Related Work

Active messages [15] invoke a receiver-side handler whenever a message arrives. Control information at the head of each message specifies the address of a user-level routine that is responsible for extracting the message from the network. This approach often requires servicing an interrupt for each message received.

The Myrinet [2] network interface has a programmable network coprocessor which implements packet-switched message passing. The coprocessor gives the network interface flexibility to implement a variety of receive-side interfaces, but the Myrinet system as shipped does not support any policies as aggressive as ours.

Several experimental network interfaces allow a sending process to deposit data directly into the memory of a receiving process [1, 3, 7, 8, 12]. Though these systems differ in several important aspects, they all provide the basic functionality to support efficient user-level communication, so they all face the problem we are trying to solve. None of them has yet provided a solution, but our solution strategy should work on all of them.

The *polling watchdog* [11] is a technique that tries to minimize the number of interrupts by delaying interrupting the receiving process in case it reads the incoming data in time and avoids needing to be interrupted. This work was done on a different architecture which requires the receiver to explicitly extract the incoming message.

7 Future Work

We plan to continue this work along the following lines:

- *scheduling effects*: investigation of the kernel scheduler's impact on the spin/blocking mechanism
- *user-level threads*: integration of this mechanism with user-level threads to obtain finer grain multiprogramming and to better overlap communication with computation
- *adaptive algorithms*: implementation of adaptive spinning strategies which update the spinning duration at runtime

- *application control*: investigation of spinning strategies that give applications direct control
- *other hardware platforms*: experimentation with this mechanism on other network interfaces, such as Memory-Channel and Myrinet

8 Conclusion

In this paper, we have studied several ways to reduce the CPU overhead required to detect data arrival in user-level communication. We focus on two key techniques to achieve low CPU overhead: combining polling with the process-blocking mechanism and reducing the overhead of servicing message-driven interrupts and of awakening sleeping processes. The two approaches led to the design of the `spin-block` mechanism that utilizes the *SHRIMP* 's message-driven interrupt facility and kernel scheduler to achieve low overhead in waiting for data arrival.

We evaluated this mechanism along with several other strategies for several workloads in a multiprogramming environment. Our experiments show that the combination of polling and blocking gives consistently good performance in a wide variety of situations, and that reducing the interrupt service time, as well as the number of unnecessary re-schedules, results in significant performance improvement.

Acknowledgments

We would like to thank Matt Blumrich, Doug Clark, Cezary Dubnicki, Liviu Iftode, Kai Li, Rob Shillner, and the rest of the *SHRIMP* Group at Princeton for their many useful suggestions that contributed to this work. We are also grateful to the anonymous referees for their helpful comments.

References

[1] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of 21th International Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[3] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. Hamlyn: A high-performance network interface with sender-based memory management. In *Proceedings of Hot Interconnects '95 Symposium*, Aug. 1995. Also available as HP Laboratories technical report HPL-95-86, July, 1995.

[4] S. Damianakis, Y. Chen, and E. W. Felten. Reducing waiting costs in user-level communication. Technical Report TR-525-96, Dept. of Computer Science, Princeton University, Sept. 1996.

[5] S. Damianakis, C. Dubnicki, and E. W. Felten. Stream sockets on shrimp. Technical Report TR-513-96, Dept. of Computer Science, Princeton University, Feb. 1996.

[6] C. Dubnicki, L. Iftode, E. W. Felten, and K. Li. Software support for virtual memory-mapped communication. In *Proceedings of the IEEE 8th International Parallel Processing Symposium*, Apr. 1996.

[7] E. W. Felten, R. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early experience with message-passing on the shrimp multicomputer. In *Proceedings of 23th International Symposium on Computer Architecture*, May 1996.

[8] R. B. Gillett. Memory channel network for PCI. *IEEE Micro*, 16(1):12–18, Feb. 1996.

[9] A. Karlin, K. Li, M. Manasse, and S. Owicki. Emprical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 41–55, Oct. 1991.

[10] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *Proceedings of 21th International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[11] O. Maquelin, G. R. Gao, H. H. Hum, K. B. Theobald, and X.-M. Tia. Polling watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of 23th International Symposium on Computer Architecture*, pages 179–188, May 1996.

[12] E. P. Markatos and M. G. H. Katevenis. Telegraphos: High-performance networking for parallel processing on workstation clusters. In *Proceedings of IEEE 2nd International Symposium on High-Performance Computer Architecture*, pages 144–153, Feb. 1996.

[13] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, Oct. 1982.

[14] R. A. Shillner and E. W. Felten. Simplifying distributed file systems using a shared logical disk. Technical Report TR-524-96, Dept. of Computer Science, Princeton University, Sept. 1996.

[15] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.