Reducing Waiting Costs in User-Level Communication
Princeton University Technical Report TR-525-96

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, 9, 19, 4, 3]. 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 copying and kernel system calls.

In this paper we address the receiving mechanism as it is the difficult operation in user-level communication architectures. First, 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.

The user-level library must either poll, or block and be awakened later by an interrupt, to find out that new data has arrived. 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 policy and which mechanism to use.

We first describe the tradeoffs between polling and blocking, drawing on previous work on spinning vs. blocking tradeoffs in multiprocessor locking. We discuss which mechanisms for blocking and unblocking are the most efficient.

In order to minimize the interrupt service time we keep the interrupt path as simple as possible. Our mechanism does not execute a user-level interrupt handler, since invoking such a handler requires too much overhead.

We present an implementation on the prototype *SHRIMP* multicomputer [1, 9], and the results of experiments using our user-level sockets library [7], 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

This section discusses the general architectural assumptions on which our work is based. We then detail the *SHRIMP* architecture on which we built our specific implementation.

2.1 Architectural Assumptions

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.

We do not make assumptions about the cost of sending data, since it is relatively independent of which policies we use. The sender's job is easier than the receiver's, because the sender knows when a message will be sent but the receiver doesn't know when a message will arrive.

2.2 The *SHRIMP* Architecture

Our implementation is built on the *SHRIMP* multicomputer. *SHRIMP* [1, 9] is a network of Pentium PCs running the Linux operating system. The network is a multicomputer routing network [17] connected to the PC nodes via custom-designed network interfaces. The *SHRIMP* network interface closely cooperates with a thin layer of software to form a communication mechanism called virtual memory-mapped communication (VMMC) [8]. This mechanism supports various message-passing packages and applications effectively, and delivers excellent performance [9].

The prototype system consists of four interconnected nodes. Each node is a DEC 560ST PC with a 60 MHz Pentium CPU [15]. The network connecting the nodes is an Intel routing backplane consisting of a two-dimensional mesh of Intel Mesh Routing Chips (iMRCs) [17], and is the same network used in the Paragon multicomputer [11]. The backplane supports deadlock-free, oblivious wormhole routing [5], and preserves the order of messages from each sender to each receiver.

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

The system's key component is the custom network interface [1, 2]. It connects each PC node to the routing backplane and implements the hardware support for VMMC.

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.

2.3 *SHRIMP* Communication Software

We have built several communication libraries for *SHRIMP*, implementing various standard communication interfaces. All of these libraries have a common structure [9] 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 [9] 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. This section describes our approaches to these goals.

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 that isn't necessary for building user-level communication libraries. In early implementations of our system, we found ourselves installing null handlers and using the notification merely to awaken a sleeping process.

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.

Although our *SHRIMP* hardware supports user-level handlers, we do not use them because they incur too high a cost for our purposes. Generally, for a network interface device driver interrupt handler to invoke a user-level handler (instead of simply setting the receive process's state to runnable), the system would have to perform the following additional work:

- the device driver must send a signal (or similar lightweight notification) to the sleeping user process;
- the user process must process the notification:
  - save the process context on the stack;
  - set up a stack frame for the handler;
  - jump to the handler;
  - restore the process state;

Just this overhead adds at least $10\mu s$ on our hardware, not including the time required for the user-level handler to do any real work. Still, we use the notification mechanism as a point of comparison with our system. Our new mechanism adds a system call that blocks the calling process until specific data arrives. The user-level process says 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. It may seem at first that we should try to minimize the number of interrupts, but this is not quite correct. Certainly we want to avoid taking an interrupt when there is no reason to do so, but there are cases in which interrupts are helpful. For example, if a process is waiting for a message and has nothing useful to do until the message arrives, and we know that the message will not arrive soon, then it is best for the process to block and wait for an interrupt to reawaken it. This gives other processes a chance to run.

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 locking, as discussed by Karlin et al [12]. 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 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 shown up yet. This policy does well in the two limiting cases of very small and very large arrival delay. Karlin et al showed [12] by a simple argument that this policy is 2-competitive: its cost is never more than twice that of the offline optimal policy, regardless

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*

of the sequence of arrival delays.

3.3 Strategies

We identified 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.

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.
- **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.

Figure 2 illustrates how these three pieces work together by showing what happens when the data arrives at various times. Our technical report [6] contains a more detailed justification of the correctness of this code.

4.1 User-level Library

All seven of the policies are implemented in `SpinBlock_Flags()`, the user-level library function. `Block_Flags()`, the system call that blocks a process, is used only when the spinning strategy ends before the data arrives.

The arguments to `SpinBlock_Flags()` are: `count` specifies the number of flags; `ptrAry`, an array of pointers to flags; and `valAry`, an array that specifies an initial value for each flag. When a flag's value is different from the one specified in `valAry`, it indicates that a suitable message has arrived. `strategy` specifies which spinning strategy to use and `timeout` can limit the time a process will sleep. `SpinBlock_Flags` returns as soon as any of the flags has changed; that is, when some `valAry[i]` has a value different than `ptrAry[i]`.

`SpinBlock_Flags` works as follows:

```
int SpinBlock_Flags (count, ptrAry, valAry,
```

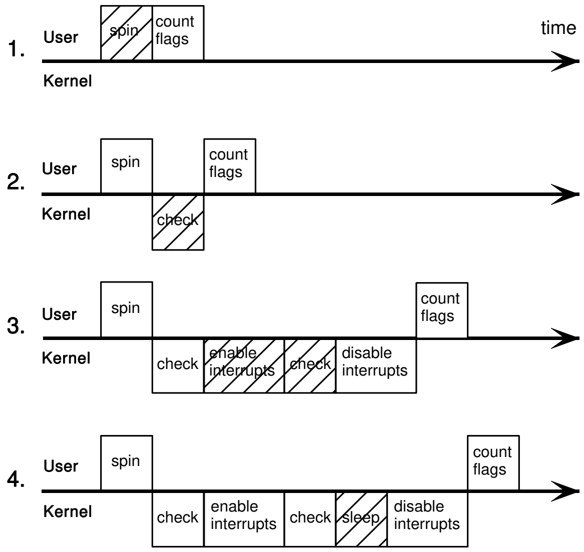

Figure 2: *Shows the four possible data arrival points (shaded) in the Spin-Blocking implementation. (1) Data arrives while in the user-level spin. (2) Data arrives during entry to the kernel; a kernel-level poll detects it. (3) Data arrives while interrupts are being enabled; a kernel-level check detects it. (4) Data arrives while the process is asleep; an interrupt handler reawakens the process.*

```
                         strategy, timeout)
{
   while(strategy says to keep spinning){
     check flags
     if(any flag changed)
       return (number of changed flags);
     if(strategy says to yield the CPU)
       yieldCPU();
   }
   return Block_Flags(count, ptrAry, valAry);
}
```

`SpinBlock_Flags()` returns a lower bound on the number of flags that have changed.

4.2 Kernel System Call

`Block_Flags()` is the kernel system call that puts a process to sleep until new data arrives. It has the same semantics as `SpinBlock_Flags`, except that it blocks the process rather than spinning.

```
    int Block_Flags (count, ptrAry, valAry)
    {
 1      for each flag pointer {
 2         enable page interrupt
 3         set page bit vector
 4         insert (ptr,val) pair in hash table
        }
 5      set process state to BLOCKED
 6      for each flag pointer
 8         if data has changed
 9            set process state to RUNNING
10      if process state is BLOCKED
11         call the process scheduler

        /* cleanup */
12      for each pointer {
13         count # of changed flags
14         disable page interrupt
15         clear page bit vector
        }
16      set process state to RUNNING
17      return count of changed flags
    }
```

Line 2 turns on the network interface's incoming interrupts, but only for those pages containing at least one of the `ptrAry` flags.[2] Line 3 is an optimization that keeps a bit vector for each page and allows the system to ignore interrupts caused by incoming messages to other locations on the same page as the flag(s). Line 4 stores the flag pointer (i.e. address) with its initial value. This is needed to detect if an incoming message changes the value of a flag. Line 5 makes the process not runable. Lines 6–9 check to see if any flags have changed, to avoid a race condition between the setting of the interrupt bits and the arrival of new data. Lines 10–11 put the process to sleep if no flags have changed.

We reach line 12 one of two ways: a) the flag checking in lines 6–9 detected a change and the process was not blocked, or b) the process was blocked on line 11 and the incoming interrupt woke up the process. Either way, we must now disable interrupts again and clean up, lines 12–15. Line 16 makes the process runnable again and line 17 returns the appropriate flag count.

4.3 Device Driver Interrupt Handler

We added a simple hook to the interrupt handler in the device driver for the *SHRIMP* network interface. When an interrupt occurs, the hook checks whether the interrupting message has changed a flag that some process is waiting for. If it has, the process is made runnable and the interrupt bits are cleared for all pages the process was waiting for. The result of this hook is

[2] Normally, the user program runs with interrupts turned off.

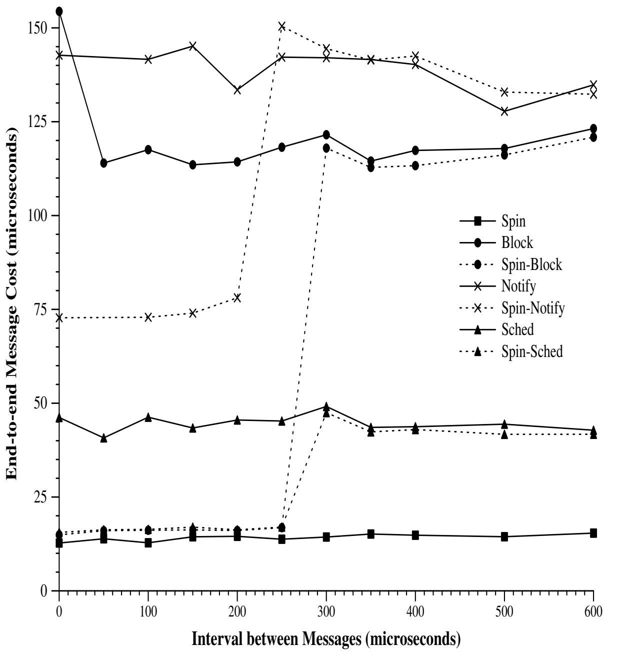

Figure 3: *Performance of a single ping-pong program for each strategy with message delay ranging from* 0 *to* $600\mu s$.

that the process reawakens whenever one of the flags changes.

Together, the three pieces of code 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.

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* [8]. 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 3 shows the performance for all of the spinning strategies when the message delay varies from 0 to $600\mu s$. Since the amount of message delay is the same for all strategies and is included in but does not reflect the time a process spends waiting for a reply message, we subtract this delay from the average time per message in our graph.

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, 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 $300\mu s$[3] before switching to the respective pure strategy. 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.

This experiment uncovered a performance bug in the Linux kernel. There was a race condition between the code that made a process runnable when a signal was delivered to it, and the code that halted the machine until the next interrupt when there were no runnable processes. The result was that the machine would sometimes be spuriously halted, and nothing would happen until the next timer interrupt, even though there was a runnable process. This bug showed up as a spike in the `notify` curve for a particular value

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

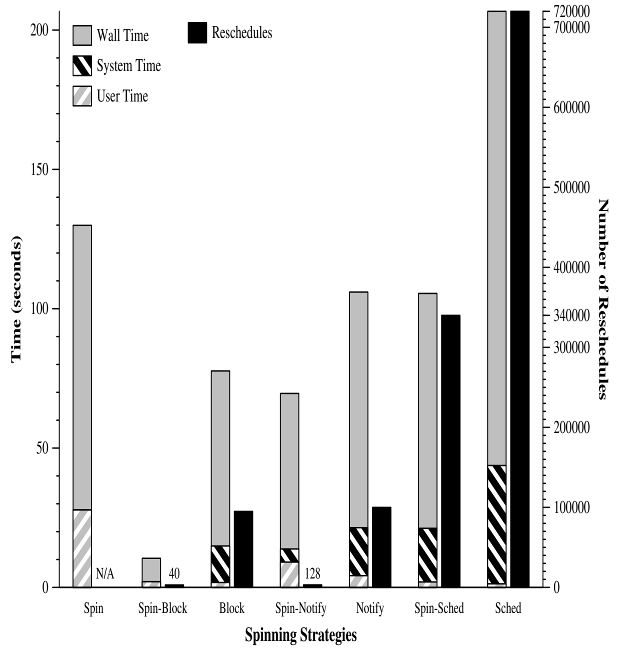

Figure 4: *Performance for 5 pairs of processes ping-ponging 100,000 messages with a random delay of* $0 - 300\ \mu s$ *between messages. The spinning strategies use a spin delay of* $300\mu s$.

of the message delay. Removing the race condition eliminated the performance surprise.

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 4 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.

Our starting point is the pure `spin` strategy. Each process spins at the user level waiting for data to arrive. This is not a good idea because each process hogs the CPU which does not allow for overlapping computation and communication. One process may be spinning uselessly while another process has a message waiting.

Clearly we should give up the CPU when we are not using it. This leads to the three basic strategies: `block`, `notify`, and `sched`. If data is not present `sched` simply gives up the CPU, `notify` puts itself to sleep, and `block` turns on interrupts and then sleeps. In Figure 4 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 is thrashing; 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 in this way, 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.

5.2 `ttcp`

`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 [7].

We ran several independent instances of each benchmark at the same time on the same pair of nodes, and summed the bandwidth reported by the instances. In an ideal world, the total bandwidth reported by k benchmarks running concurrently would be the same bandwidth reported by a single benchmark running alone. The results of our experiment are shown in Figure 5.

With one instance of `ttcp`, `spin` performs best since there is no reason to ever block with only one process per node. As the number of instances increases though, `spin`'s performance degrades quickly

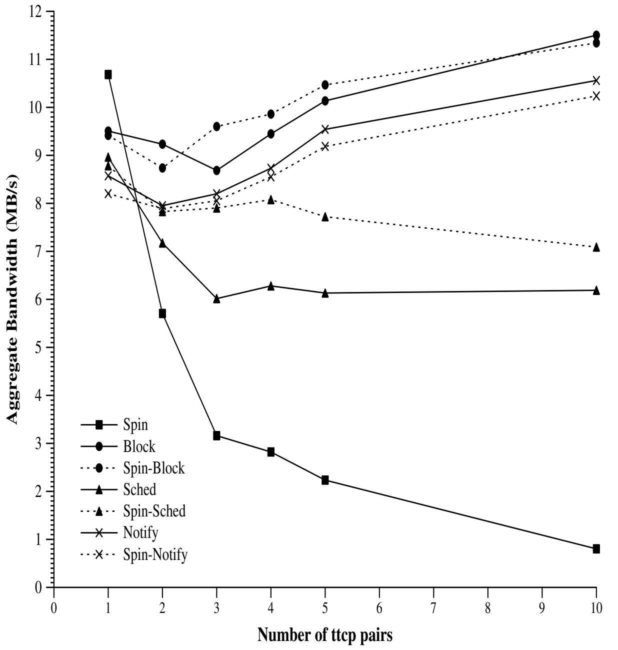

Figure 5: *Performance of several concurrent instances of* `ttcp`*. The graph shows the total bandwidth achieved as a function of the number of ttcp instances being run concurrently.*

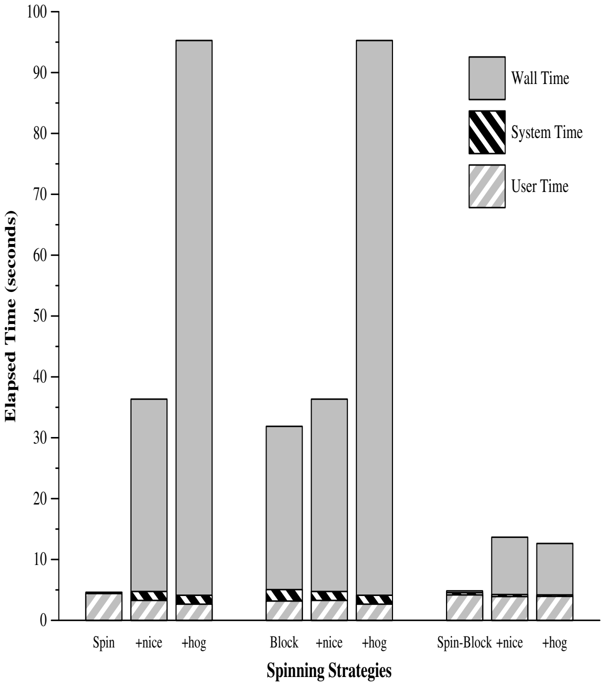

Figure 6: *Performance of a distributed file system benchmark.*

as it becomes less and less likely that communicating processes are co-scheduled. `block` exhibits fairly constant bandwidth; it is the right strategy in this case since the message delay is large due to the sheer size of the messages being transferred[4]. As usual, `spin-block`performs roughly the same as the better of `spin` and `block` (in this case `block`). `sched` and `notify`, along with their spin hybrids, perform as expected: better than `spin` but worse than `block` and `spin-block`.

5.3 Distributed File System

Our next set of experiments uses a distributed file system [16] 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.

[4]Each instance transfered 2,048 8KB messages

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 three policies that had a chance of having the best performance: `spin`, `block`, and `spin-block`.

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 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 6. 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 3). Blocking in this case would just add 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 context switched before the message arrives, it has to wait for a full scheduling quantum before it can check for the data. Meanwhile, the file system process on the other node would either exhaust its scheduling quantum, or decides to use `block` strategy and goes to sleep. Hence, a vicious cycle of context switch thrashing would arise. This explains why both `spin` and `block` strategies perform badly in the presense 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 get out, of a thrashing cycle. This results in the excellent performance of `spin-block` with both `hog` and `nice` processes.

6 Related Work

Active messages [18] 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 [3] network interface has a network coprocessor which implements packet switching 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 proces [1, 4, 9, 10, 14, 19]. 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* [13] is 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.

Karlin et al [12] studied the effects of spinning vs. blocking tradeoffs on multiprocessor performance in the context of locks. We applied many of their ideas in our work.

7 Future Work

We plan to continue this work with the following ideas:

- *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*: we intend to implement more spinning strategies. For example, adaptive algorithms that update the spinning duration at runtime.
- *application control*: investigate spinning strategies that give applications direct control.
- *other hardware platforms*: port this mechanism to other network interfaces, namely MemoryChannel 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 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.

This project is sponsored in part by ARPA under contract N00014-95-1-1144, by NSF under grant MIP-9420653, by Digital Equipment Corporation and by Intel Corporation. Felten is supported by an NSF National Young Investigator Award.

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, April 1994.

[2] Matthias A. Blumrich. *Network Interface for Protected, User-Level Communication.* PhD thesis, Dept. of Computer Science, Princeton University, June 1996. Available as technical report TR-522-96.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[4] Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes. Hamlyn: A High-Performance Network Interface with Sender-Based Memory Management. In *Proceedings of Hot Interconnects '95 Symposium*, August 1995. Also available as HP Laboratories technical report HPL-95-86, July, 1995.

[5] William J. Dally and Charles L. Seitz. Deadlock Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[6] Stefanos Damianakis, Yuqun Chen, and Edward W. Felten. Reducing Waiting Costs in User-Level Communication. Technical Report TR-525-96, Dept. of Computer Science, Princeton University, September 1996.

[7] Stefanos Damianakis, Cezary Dubnicki, and Edward W. Felten. Stream Sockets on SHRIMP. Technical Report TR-513-96, Dept. of Computer Science, Princeton University, February 1996.

[8] Cezary Dubnicki, Liviu Iftode, Edward W. Felten, and Kai Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the IEEE 8th International Parallel Processing Symposium*, April 1996.

[9] Edward W. Felten, Richard Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos Damianakis, Cezary Dubnicki, Liviu Iftode, and Kai Li. Early Experience with Message-Passing on the Shrimp Multicomputer. In *Proceedings of 23th International Symposium on Computer Architecture*, May 1996.

[10] Richard B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[11] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[12] A.R. Karlin, Kai Li, M.S. 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, October 1991.

[13] Olivier Maquelin, Guang R. Gao, Herbert H.J. Hum, Kevin B. Theobald, and Xin-Min 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.

[14] Evangelos P. Markatos and Manolis 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, February 1996.

[15] Avtar Saini. An Overview of the Intel Pentium Processor. In *Proceedings of the IEEE COMPCON'93 Conference*, pages 60–62, February 1993.

[16] Robert A. Shillner and Edward W. Felten. Simplifying Distributed File Systems Using a Shared Logical Disk. Technical Report TR-524-96, Dept. of Computer Science, Princeton University, September 1996. Submitted for Publication.

[17] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.

[18] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages:

A Mechanism for Integrated Communication and Computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[19] John Wilkes. Hamlyn—An Interface for Sender-Based Communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, November 1992.