Efficient Sleep/Wake-up Protocols for User-Level IPC

Ronald C. Unrau
Department of Computing Science
University of Alberta, CANADA
unrau@cs.ualberta.ca

Orran Krieger
IBM T.J. Watson Research Centre
Yorktown Heights, New York
okrieg@watson.ibm.com

December 9, 1997

Abstract

We present a new facility for cross-address space IPC that exploits queues in memory shared between the client and server address space. The facility employs only widely available operating system mechanisms, and is hence easily portable to different commercial operating systems. It incorporates blocking semantics to avoid wasting processor cycles, and still achieves almost twice the throughput of the native kernel-mediated IPC facilities on SGI and IBM uniprocessors. In addition, we demonstrate significantly higher performance gains on an SGI multiprocessor. We argue that co-operating tasks will be better served if the operating system is aware of the co-operation, and propose an interface for a hand-off scheduling mechanism. Finally, we report initial performance results from a Linux implementation of our proposal.

1 Introduction

Inter-Process Communication (IPC) is a basic service provided by all multi-tasking operating systems. IPC is an integral part of concurrent applications such as client-server environments (eg. windowing systems or data bases); IPC is also integral to parallel applications that must co-ordinate worker activities (eg. using barrier operations or task queues). There are also several reasons to use kernel-level processes rather than light-weight threads: they are necessary if the processes reside in different address spaces; and the application can take advantage of kernel-level scheduling, pre-emption, priority management, etc. User-level IPC uses memory segments shared within or across address spaces as the basis for implementing a communication protocol. User-level IPC is attractive for several reasons: first, the number of (expensive) system calls can be reduced; second, the amount of copying into and out of the kernel address space is reduced; and third, custom protocols that take advantage of application-specific knowledge can be used. To obtain the best overall system throughput, particularly in multi-programmed environments, the IPC mechanism should support blocking semantics. Blocking IPC semantics dictate that a process should sleep if the other processes with which it wishes to communicate are not ready; when those processes become ready, they should wake-up their sleeping partners. In this paper, we present and evaluate several protocols that use shared FIFO queues to efficiently implement a user-level client-server communication substrate with blocking semantics.

Under ideal circumstances, user-level IPC can achieve round-trip latencies on the order of tens of instructions. The ideal situation is when a message transfer is achieved by simply adding and/or removing requests from queues in shared memory; that is, when there is no need to invoke kernel services. On a shared-memory multiprocessor, if the client and server are running on different processors (and are willing to spin for a time) this efficient situation may be common. On a uniprocessor, this efficient situation can be common if the IPC is asynchronous. In this case a client process can enqueue multiple asynchronous messages on to a shared queue without blocking waiting for a response. Similarly, when the server gets the opportunity to run, it can handle requests and respond without invoking kernel services until all pending requests are processed.

The challenge for user-level IPC is to handle synchronous message passing efficiently, especially on a uniprocessor. Synchronous message-passing typically results in at most one or two messages in a queue, and an empty queue is actually the common case. If a queue is empty (or full), the simplest solution is to busy-wait until until some data arrives (or until there is room for more data). This greedy solution attempts to maximize performance by minimizing latency. However, the performance is gained at the cost of reduced overall system throughput, which is lost in several ways. First, if client messages are relatively infrequent the server wastes resources by spinning when no work is available. The clients can waste resources while busy-waiting for their reply, which can be significant if the server, for example, is performing I/O to a disk or network on the clients behalf. In multi-client environments, clients compete against each other and with the server, whose processing time is impeded by unnecessary context switches and the subsequent loss of cache state, etc. We should also point out that the impact of busy-waiting is substantially higher on uniprocessor than on multiprocessor systems, although the throughput arguments apply equally to both. Some compromise can be achieved by periodically polling the queue for work, but response time suffers if work is enqueued just after the queue has been sampled. Exponential back-off sampling as used for spin locks is not appropriate because the queue is not necessarily contended, although the back off sampling does in some sense model a Poisson arrival expectancy. It is not cost effective to have a server or client busy waiting for work on a multiprocessor system, and unacceptable on a uniprocessor.

In this paper, we present three new protocols that can be used to incorporate blocking semantics into user-level IPC. The protocols are presented in the context of a Send/Receive/Reply interface layered over shared-memory based unidirectional concurrent queues. We first develop a basic functional blocking user-level IPC protocol (Section 3), but we show that because of interactions with the operating system scheduling policy, the protocol has worse performance than standard kernel-mediated IPC. We then give two enhancements to the protocol that improve performance for commercial operating systems (Section 4), including multiprocessor systems. Finally, we argue in Section 6 that the scheduling policy requirements for user-level IPC are fundamentally different

```
void Send( Msg *msg, Msg *ans ) {
  while( !enqueue( Q[srv], msg ) )
    busy_wait(); /* queue full */

  while( !dequeue( Q[clnt], ans ) )
    busy_wait(); /* wait for reply */
}
```

```
void Receive( Msg *msg ) {
  while( !dequeue( Q[srv], msg ) )
    busy_wait(); /* queue empty */
}

void Reply( int clnt, Msg *msg ) {
  while( !enqueue( Q[clnt], msg )
    busy_wait(); /* queue full */
}
```

Figure 1: **Both Sides Spin**: A Send/Receive/Reply interface with fixed sized messages and busy waiting using shared memory queues.

than for typical multi-tasking workloads, and suggest an extended kernel interface to support our hand-off scheduling policy. We show the results of this policy when implemented in the Linux operating system.

The purpose of this paper is to examine the performance issues associated with user-level IPC, and due to space limitations we do not deal with security issues. Servers can protect themselves from clients by careful access to the shared memory queues. Clients can be protected from other clients by placing only recoverable control information in the queues shared by other clients; all sensitive state is maintained in mapped regions isolated from the other clients.

2 IPC Primitives

We start by presenting and examining a simple user-level IPC mechanism based on concurrent uni-directional queues implemented in shared memory. Using a Send/Receive/Reply interface and a busy-wait implementation, we can explore the factors affecting performance for subsequent user-level IPC algorithms that include sleep/wake-up.

2.1 Busy-Waiting User-Level IPC

Concurrent queues are useful in many applications, for example, to implement a task queue in a parallel program. Here we are concerned with their application to user-level IPC in a client-server environment. From this perspective, concurrent queues can be considered as FIFO flow-controlled half-duplex streams similar to UNIX pipes. Concurrent queues can serve as a basis for communications protocols that support multiple clients and multiple server threads. We chose to layer a Send/Receive/Reply interface over the base enqueue/dequeue interface as shown in Figure 1.

The figure shows a synchronous Send and the corresponding Receive/Reply implemented using busy-wait over the base queue interface[1]. We shall refer to this basic implementation as **Both Sides Spin (BSS)**. The interface uses fixed sized messages to permit efficient free-pool management. Variable sized messages can be accommodated by using one of the fields of the fixed sized message to point to a variable sized component in shared memory.

[1]The interface is easily augmented to support asynchronous Sends, in that they are functionally identical to Reply.

The implementation in Figure 1 uses two queues: a receive queue at the server for incoming messages, and a reply queue for responses back to the client. If multiple clients want to connect to the server, the single receive queue is still adequate but a reply queue per client is required. In this case, each client request should include the number of the reply queue to be used for the response. This server architecture is used for the performance evaluation of our sleep/wake-up protocols; an alternative architecture might be to have a server thread per client, but that would require two queues per client to implement the full-duplex virtual connection.

The `busy_wait` functions of Figure 1 can be implemented in any number of ways, including a null function. Note that while a null `busy_wait` will minimize latency on a multiprocessor (albeit at the expense of other concurrently executing processes), the same function will degrade performance on a uniprocessor. On uniprocessors `busy_wait` should be implemented as a `yield()` system call to allow other processes to progress - otherwise the client or server will spin until its quantum expires.

2.2 Performance

In subsequent sections we show how sleep/wake-up can be incorporated into the basic BSS algorithm. However, it is important to understand the performance of the base algorithm, since it represents an upper bound on performance. In this section we analyze the performance of the **BSS** algorithm on two different commercial operating systems.

To evaluate the protocol, we implemented a simple client-server system where up to n clients connect to a single-threaded server and make requests using our user-level IPC interface. The clients connect to the server, barrier, and then enter a tight loop where they barrage the server with many thousands of message requests. Each message contains 24 bytes which include: an opcode to identify the request type; the channel on which to return the result; and a double precision floating point value that serves as an argument to the request. The server is placed in a tight Receive/Reply loop that accepts connections and processes requests, where the processing per request is simply to echo the argument back to the client. The server throughputs are calculated using the real elapsed time from the first message request (to exclude the connect time processing) until the last client disconnects (because the server does not know in advance how many messages it must process).

The evaluation software uses a common implementation of the Michael and Scott two-lock queue [9], and a common client and server implementation. Between all the different results presented in this and later sections, only the implementation of the protocols themselves changes. Also, the software is identical between the uniprocessor and multiprocessor environments, except that busy-waiting is implemented as a `yield()` system call on the uniprocessor and as a busy-wait delay loop on the multiprocessor.

Figure 2 shows the measured server throughput (ie. larger numbers are better) in messages per millisecond for 1 to 6 clients on two different uniprocessor systems. The results in the left graph of the figure were obtained from an SGI Indy running IRIX 6.2 on a 133 MHz MIPS R4000 processor; the results on the right are from an IBM P4 running AIX 4.1 on a 133 MHz PowerPC 604 processor. Both systems are configured with a 32 KByte split L1 cache, and a 512 KByte combined L2 cache.

Each graph in Figure 2 contains two curves. The upper curve is the measured server throughput for the Both Sides Spin algorithm of Figure 1. For comparison purposes, we also show the measured server throughput using System V message queues. As a kernel mediated IPC mecha-

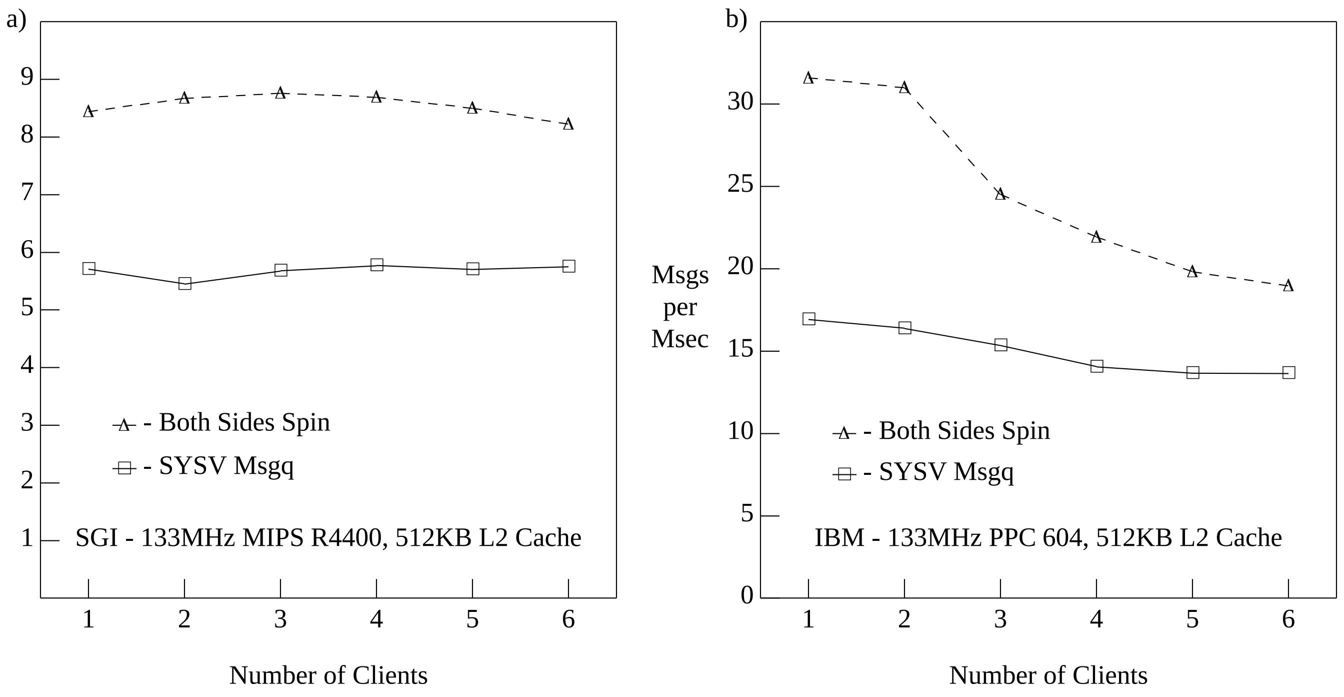

Figure 2: Measured uniprocessor Server Throughput in Messages/millisecond for varying numbers of client processes. The left and right graphs are measured throughputs from 133MHz SGI and IBM machines, respectively. The top curve in each plot is the throughput for the BSS algorithm; the lower curve shows the throughput when SYSV message queues are used.

nism, SYSV message queues represent a lower-bound on acceptable user-level IPC performance. The performance is a lower bound in the sense that while there may be many advantages to using shared-memory based communication, performance is usually the primary motivator.

Concentrating for the moment on the SGI performance results (Figure 2a), we see that the BSS curve exhibits the non-intuitive effect of *increasing* throughput as the number of clients increases. To understand this behavior, consider first the execution interleaving when a single client and server exchange messages. From the BSS algorithm of Figure 1, the client enqueues its request to the server, and then, because Sends are synchronous, the client immediately tries to dequeue the reply. On a uniprocessor, the reply queue is usually empty (because the server has not had a chance to run and enqueue the reply) so the client yields the processor. This allows the server a chance to run: it dequeues the request; enqueues the reply; and immediately attempts to receive the next request. Again, there can be no request because the client has not had a chance to run, so the server yields the processor.

When multiple clients run simultaneously, one would intuitively expect server throughput to *decrease*, because many processes are actively wasting processor cycles context switching and waiting for their request to complete. Instead, we see server throughput *increase*, because it turns out the overall number of context switches is actually reduced. The reason is that the server can avoid context switches if there are multiple messages on its input queue when the server is given its time slice. This behavior was confirmed using the `getrusage` system call to obtain the number of voluntary and involuntary context switches. The analysis showed that for, say, 100000 requests from a single client the server made 100000 voluntary context switches; with 2 clients each making

Primitive Operation		Machine SGI	IBM
`enqueue/dequeue`		3 μsec	2 μsec
`msgsnd/msgrcv`		37 μsec	15 μsec
Concurrent Yields	1 process	16 μsec	6 μsec
	2 processes	18 μsec	13 μsec
	4 processes	45 μsec	30 μsec

Table 1: Measured times for primitive operations.

100000 requests `getrusage` still reported only 100000 voluntary server context switches.

Turning now to the IBM performance results (Figure 2b), we see completely opposite performance trends compared to the SGI system as the number of clients increase. Instead of increased throughput as the number of clients increases, the throughput rolls off from a high of 32 messages/msec (for BSS) to 19 messages/msec with 6 clients. It is also interesting to see that the performance of System V IPC does not roll off as quickly as the user-level IPC protocol.

Considered together, the two graphs show that even this simple user-level IPC algorithm is heavily influenced by system-level scheduling policies. From Figure 2, one can see that the throughput of user-level IPC outperforms kernel-mediated IPC by factors of more than 1.5 and 1.8 for the SGI and IBM systems, respectively. While this performance is good, we expected it to be better. From our earlier discussion, the BSS algorithm should involve two system calls (`yield`) and two context switches per message exchange. For SYSV message queues, we expect four system calls (a `msgsnd/msgrcv` pair at both the client and the server) and two context switches per message exchange. Further, we expect the cost of a `yield` system call to be much cheaper than either `msgsnd` or `msgrcv`.

To investigate further, we first measured the times for the primitives involved in the experiment. The results are summarized in Table 1. The `enqueue/dequeue` and `msgsnd/msgrcv` times are for the combined pair of operations as executed by a single process running in a tight loop. To measure concurrent yield times, n clients would first barrier and then all processes entered a tight `yield` loop. The times reported are the average loop trip time per process.

From Figure 2a, the round-trip latency on the SGI machine is about 119 μsec when there is 1 client. Since a round-trip involves 2 `enqueue/dequeue` pairs and 2 context switches, Table 1 suggests a round-trip latency of $2 \times 3 + 2 \times 18 = 42$ μsec – less than half of our observed latency! Instrumentation of the code revealed that each process on the SGI was performing approximately 2.5 yields per round-trip message exchange. This suggests that the degrading priority scheme used by the operating system for scheduling is preventing the process that just enqueued a message from yielding the CPU to the waiting process. Note that since the both the client and server are busy-waiting, the operating system always sees both processes as ready; it is only after the active process has accumulated sufficient execution time that its priority is degraded enough to warrant a full context switch.

To test the hypothesis that priority aging by the operating system is impacting performance, we set both the server and client priorities to be non-degrading. Figure 3 shows the results as server time in microseconds per message (ie. lower numbers are better). The curves show that the throughput is increased by 50% on the SGIs, and 30% on the IBMs. We also reran the Sys-

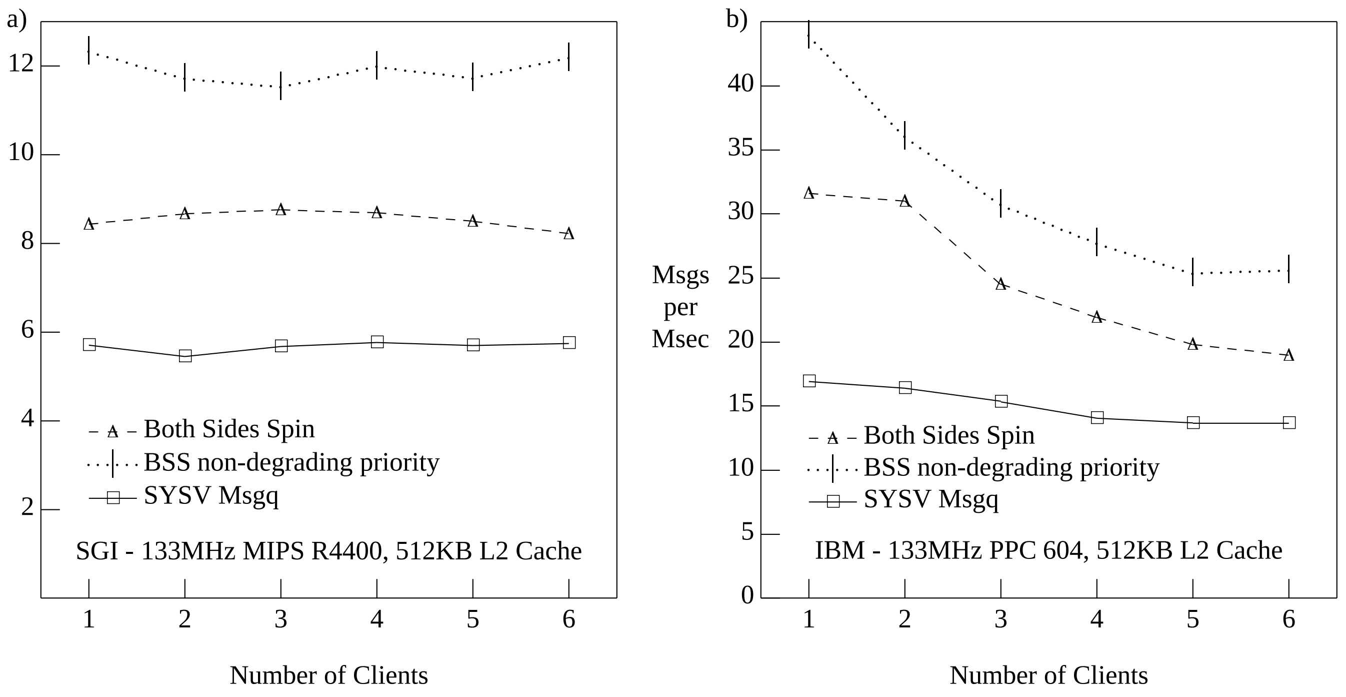

Figure 3: Measured uniprocessor Server throughput in Messages/Millisecond for varying numbers of client processes. The bottom two curves are the same as from Figure 2; the top curve is the throughput when non-degrading priorities are used.

tem V Message Queue implementation with fixed priority scheduling, and found no appreciable difference in performance.

Unfortunately, running with fixed-priority is not practical, except possibly for real-time applications. On both the SGI and the IBM, the system call to set fixed-priority scheduling requires super-user privileges – primarily because the system can be deadlocked by careless use of non-degrading priority scheduling. Of course, the problem is not with the priorities *per sé*, the problem is that even though the CPU is explicitly `yield`ed, there is no guarantee that any other process will run. This behavior suggests a fundamentally different scheduling approach is needed for user-level IPC. The reason is that schedulers normally view all ready processes as *competing* for the CPU, however, processes involved in user-level IPC are *co-operating* towards a common goal. What we desire is a user-level call to effect hand-off scheduling from the client to the server or *vice versa*. We explore this possibility further in Section 6.

3 Incorporating Sleep/Wake-Up

In this section we show how the busy-wait implementation of user-level IPC presented in Figure 1 can be extended to incorporate blocking. Ideally, a protocol is needed such that a consumer process that finds its queue empty puts itself to sleep until there is work to do. A producer process can enqueue its work and then wake-up the consumer so that response latency is minimized. This type of condition or event synchronization is common in thread libraries that multiplex multiple light-weight threads onto a single heavy-weight (or kernel) process. We are concerned, however, with the case where multiple kernel-level processes (usually in different address spaces) must communicate. In this case, the operating system must be involved in the sleep/wake-up part of the protocol, which is not the case for light-weight thread packages. Since kernel-mediated IPC can be accomplished in only 4 system calls per round-trip message (ie. client send, server receive, server reply, client receive), any sleep/wake-up protocol must avoid unnecessary system calls and even extraneous context switches to be efficient.

The top of Figure 4 shows how sleep/wake-up (denoted in the figure as `block/unblock`, respectively) can be incorporated around each enqueue/dequeue operation. In particular, the server sleeps if there are no client requests outstanding and the clients need not compete for resources if they sleep while waiting for their reply. The difficulty in incorporating sleep/wake-up is that the consumer[2] should only block if the queue is empty, and therefore the producer must determine if the consumer should be woken when new work has been enqueued. It is important that the consumer only block if the queue is empty and that the producer only wake the consumer if it is sleeping, because sleep and wake-up are system calls. If these system calls are made for every enqueue/dequeue then the performance advantage of user-level IPC is lost. In the best case user-level IPC requires no system calls, which happens if the server and client never see their respective queues empty. In the worst case, blocking user-level IPC may need four system calls: the client sends and wakes up server, then sleeps waiting for the reply; the server processes the request and wakes up the client, then blocks waiting for the next request. The challenge is to find and exploit situations where the client and/or server need not explicitly block – this saves 2 system calls because no wake-up is needed if no sleep was performed.

A further complexity in designing a sleep/wake-up protocol is that two "incompatible" operations must be performed atomically. The consumer, on detecting an empty queue: (1) indicates that it *may* be going to sleep (step **C.2** in Figure 4); (2) executes the kernel call to do so (step **C.4** in the figure). These two steps cannot be made atomic using a lock, since the consumer cannot release the lock once it is asleep. Consequently, the sleep/wake-up protocols presented here can have race conditions. Race conditions are not necessarily harmful, in the sense of causing incorrect behavior. However, they may have an impact on performance if left unchecked. Conversely, the work-arounds to prevent race conditions can also degrade performance. As a result, race conditions that do not affect correctness are often tolerated if they are deemed to be relatively infrequent and if preventing them would unduly lengthen the execution time of the critical path.

There are (at least) three possible race conditions with the protocol given in Figure 4. The race conditions are also depicted in the figure using Execution Interleaving time-lines. These charts show execution steps progressing downwards for a consumer process and one or two producer processes: an empty step indicates that the particular process is not executing; multiple steps on the same line indicate that the operations are performed concurrently.

In the first execution interleaving scenario of Figure 4, the producer attempts to wake up the consumer after it clears the `awake` flag and before it goes to sleep. This race condition is harmful if the wake-up condition does not remain pending on the consumer, because the consumer could sleep forever. One way to ensure the condition remains pending is to implement the sleep and wake-up using counting semaphores[3].

[2]Note that the consumer is the process that is dequeuing messages, and could be either the server receiving a request or a client obtaining its reply.

[3]A *down* or `P` operation on a counting semaphore attempts to decrement a count value - the operation blocks the calling process if the count value is zero or less. An *up* or `V` operation increments the semaphore count, unblocking any waiting processes if the count value exceeds 0.

Producer/Consumer Algorithm with Blocking

```
Consumer Steps:                           Producer Steps:
  C.1 if dequeue( msg ) = EMPTY             P.1 enqueue( msg )
  C.2    consumer->awake ← FALSE            P.2 if not consumer->awake
  C.3    if dequeue( msg ) = EMPTY          P.3    unblock( consumer )
  C.4      block( consumer )
  C.5    consumer->awake ← TRUE
```

Execution Interleaving 1: Wake-up before sleep

```
Consumer:                                 Producer:
  C.1 if dequeue( msg ) = EMPTY
  C.2    consumer->awake ← FALSE
  C.3    if dequeue( msg ) = EMPTY
                                            P.1 enqueue( msg )
                                            P.2 if not consumer->awake
                                            P.3    unblock( consumer )
  C.4      block( consumer )
```

Execution Interleaving 2: Multiple wake-ups

```
Consumer:                           Producer 1:                         Producer 2:
  C.1 if dequeue(msg) = EMPTY
  C.2    consumer->awake ← FALSE
  C.3    if dequeue(msg)= EMPTY
  C.4      block( consumer )          P.1 enqueue( msg )                  P.1 enqueue( msg )
                                      P.2 if not consumer->awake          P.2 if not consumer->awake
                                      P.3    unblock( consumer )          P.3    unblock( consumer )
```

Execution Interleaving 3: Wake-up without sleep

```
Consumer:                                 Producer:
  C.1 if dequeue( msg ) = EMPTY
  C.2    consumer->awake ← FALSE            P.1 enqueue( msg )
  C.3    if dequeue( msg ) = EMPTY          P.2 if not consumer->awake
                                            P.3    unblock( consumer )
  C.5    consumer->awake ← TRUE
```

Execution Interleaving 4: Why step C.3 is required

```
Consumer:                                 Producer:
  C.1 if dequeue( msg ) = EMPTY
                                            P.1 enqueue( msg )
                                            P.2 if not consumer->awake
  C.2    consumer->awake ← FALSE
  C.4      block( consumer )
```

Figure 4: Producer/Consumer with sleep/wake-up protocol and some possible races.

A second race condition that can occur is depicted in Execution Interleaving 2 of Figure 4. Here multiple producers simultaneously see that the consumer is sleeping, and simultaneously try to wake the consumer. If counting semaphores are used as the sleep/wake-up mechanism, the multiple wake-ups can result in a semaphore value greater than zero, which means the next semaphore “down” operation performed by the consumer will simply decrement the count and not actually block. This race condition is not necessarily harmful: if the dequeue attempts are placed in a loop the consumer will simply iterate until the semaphore count reaches zero and then block. Performance is degraded because the multiple wake-ups were unnecessary and because the consumer must iterate to restore the count. Unfortunately, the race condition can be harmful if the consumer is busy enough that it never gets a chance to iterate the semaphore count down to zero. In this situation, the multiple wake-ups can accumulate - eventually causing an overflow of the semaphore value (this happened in our first version of the algorithm!). The race condition can be removed by having the producers reset the `awake` flag atomically. The flag could be protected by a lock, but a simple test and set operation is an efficient way to make sure only the first producer to see the `awake` flag cleared to 0 will reset it to 1.

Execution Interleaving scenario 3 of Figure 4 is similar in many ways to scenario 2. Here, the semaphore value can accumulate over time if the producer tries to wake a consumer that did not have to sleep because the second dequeue attempt (step C.3 of the protocol) succeeded. This can be detected if the consumer uses test and set to reset the `awake` flag if step C.3 does not find the queue empty. If the flag was already set (ie. the result of the test and set is 1) then a producer tried to wake the consumer, and the counting semaphore can safely be decremented without blocking.

The fourth interleaving scenario of Figure 4 is included to illustrate why the seemingly redundant dequeue of step **C.3** of the protocol must be included. Here the producer checks the `awake` flag after the consumer has found the queue to be empty but before the consumer can clear its `awake` flag. As a result, the producer does not attempt to wake-up the consumer, who (potentially) sleeps forever.

Given the protocol description and considerations to reduce the impact of race conditions, Figure 5 shows how the basic busy-wait IPC implementation can be extended to incorporate sleep and wake-up. We shall refer to this algorithm as **Both Sides Wait (BSW)**.

The reader should note that the `sleep( 1 )` call on a queue full condition (after enqueuing in both `Send` and `Reply`) is assumed to use the standard UNIX semantics, ie. the process will sleep for at least one second. Although this may seem a rather long sleep time, the queue full condition seldom occurs and the implication is that the consumer is saturated; waiting a full second should allow the consumer to reduce the backlog of outstanding messages.

3.1 Initial Performance

The performance of the BSW algorithm is plotted with respect to the BSS and System V message queues in Figure 6. The performance more or less matches the performance of kernel mediated IPC. This could be considered positive, since if user-level IPC is as good as kernel mediated IPC, and has advantages for asynchronous IPC, specialized protocols, and multiprocessors, then it is a win overall. However, we do not consider this sufficiently good since synchronous IPC on uniprocessors is the most important workload today.

The reason for the poor performance is that the `V` operation that wakes up the consumer does not force a rescheduling decision. Thus, if the server is blocked when the client places something

```
void Send( Msg *msg, Msg *ans ) {
  while( !enqueue( Q[srv], msg ) )
    sleep( 1 ); /* queue full */

  if( !tas( &(Q[srv]->awake) ) )
    V( srv ); /* Wake-up server */

  /* Wait for reply */
  while( !dequeue(Q[clnt],ans) ) {
    Q[clnt]->awake = 0;
    if( !dequeue(Q[clnt],ans) ) {
      P( clnt ); /* Wait for server */
      Q[clnt]->awake = 1;
    }
    else { /* reply ready */
      if( tas( &(Q[clnt]->awake) ) )
        P( clnt ); /* Fix race condition */
      break;
    }
  } /* end while */
}
```

```
void Receive( Msg *msg ) {
  while( !dequeue( Q[srv], msg ) ) {
    Q[srv]->awake = 0;
    while( !dequeue( Q[srv], msg ) ) {
      P( srv ); /* Wait for client */
      Q[srv]->awake = 1;
    }
    else { /* not empty */
      if( tas( &(Q[srv]->awake) ) ) P(srv);
      break;
    }
  } /* end while */
}

void Reply( int clnt, Msg *msg ) {
  while( !enqueue( clnt, msg ) )
    sleep( 1 ); /* queue full */

  if( !tas( &(Q[clnt]->awake) ) )
    V( clnt ); /* Wake-up client */
}
```

Figure 5: **Both Sides Wait**: A Send/Receive/Reply interface that uses counting semaphores to incorporate sleep/wake-up.

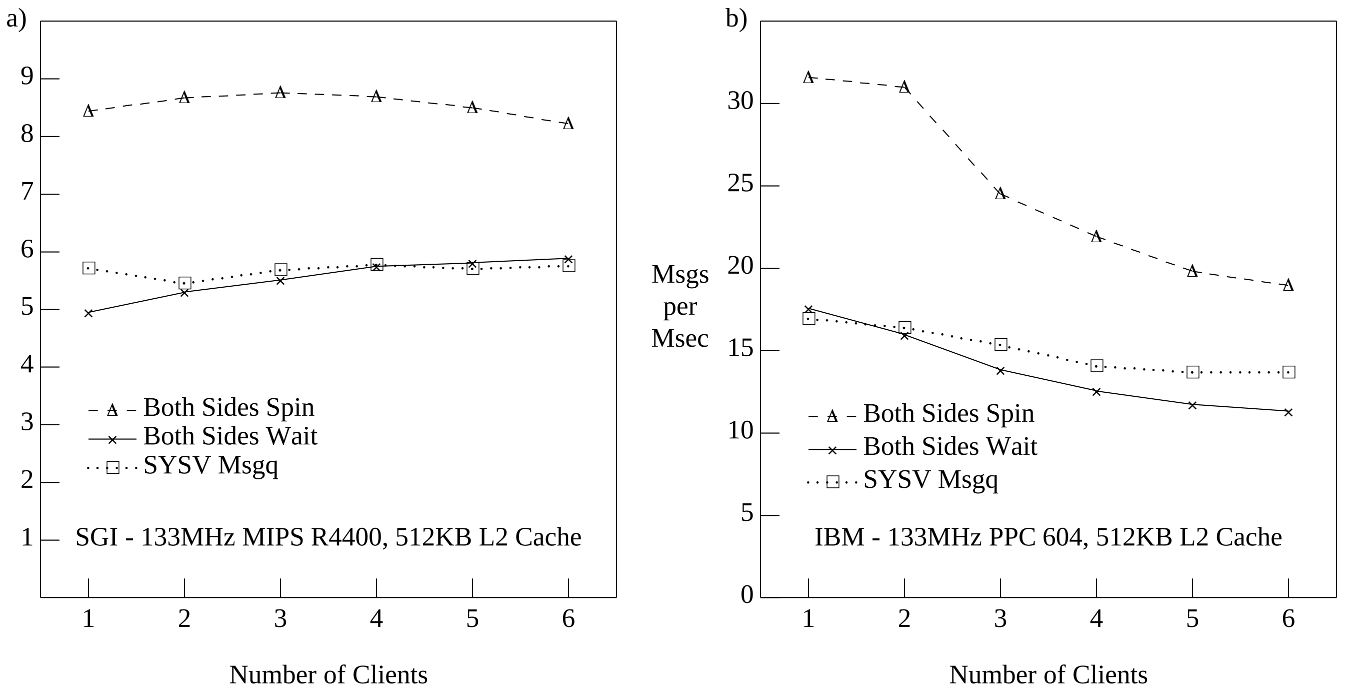

Figure 6: Measured uniprocessor Server Throughput in Messages/millisecond for varying numbers of client processes. The solid curve in each figure represents the throughput for the **Both Sides Wait** algorithm.

in the queue, the client "ups" the semaphore but then must block on the reply. At this point, the operating system can restart the server, who enqueues the reply and "ups" the client semaphore. The server then blocks waiting for the next request. The result is four system calls per round-trip: two `V` operations and two `P` operations. Since we used System V semaphores, which are of similar weight to the four System V message queue calls, there is no advantage to the shared memory solution at all.

4 Performance Improvements

The algorithms of Figure 5 achieve our goal of blocking processes that try to dequeue from an empty queue. However, all the performance advantages of user-level IPC are lost.

One could argue that if an asynchronous Send were used then useful work could be inserted between the enqueue for the request and the subsequent dequeue for the reply (assuming a reply is needed at all). Delaying the client dequeue is effective in a multiprocessor environment, especially if useful work can be accomplished. However, many systems prevent this by layering a synchronous Remote Procedure Call (RPC) interface over the user-level IPC, and it is often the case that the client actually needs the reply before it can proceed. Delaying the dequeue doesn't help at all on a uniprocessor, unless the intervening work includes a system call or is time consuming enough to expire the current quantum. The reason is that the client has to be pre-empted and the server scheduled to run for it to be able to prepare a reply. Only then does the protocol avoid a system call that would have been present using kernel mediated IPC.

4.1 Simulating Hand-off Scheduling

What we desire is a hand-off scheduling policy such that when the client executes the system call to wake the server, the server is readied and run immediately instead of returning to the client. Assuming the server can process the request in a single quantum, it would not block until it tried to dequeue the next request. At that time, the client (who was blocked as part of the wake-up system call) could continue, find its reply already available, and continue processing. Adding up the costs, we see that hand-off scheduling reduces the number of system calls from 4 to 2 per round-trip.

Figure 7 shows the **BSW** algorithm of Figure 5 modified to include `busy_wait/yield` calls to suggest hand-off scheduling to the operating system. On the client side, the `busy_wait` calls are added after the client returns from waking the server, and after the client first finds the reply queue is empty. Note that we used `busy_wait` instead of `yield` directly, so that the algorithm ports transparently between multiprocessor and uniprocessor implementations. On both architectures the `V( srv )` call readies the server. However, on a multiprocessor the server will likely execute on a different processor (assuming one is available), while on a uniprocessor the semaphore operation does not force a rescheduling decision and the client continues. Thus, on a multiprocessor the `busy_wait` will delay the client while the server processes the request, so that the reply can be ready when the client attempts to dequeue it. On uniprocessors where `busy_wait` is implemented as `yield`, the operating system is forced to at least re-evaluate whether the client or server should run.

Since the wake-up operation is only executed if the server is already blocked, the second `busy_wait` is executed if the client finds the reply queue is still empty. In this case, the `busy_wait`

```
void Send( Msg *msg, Msg *ans ) {
 while( !enqueue( Q[srv],msg ) )
   sleep( 1 ); /* queue full */

 if( !tas( &(Q[srv]->awake) ) ) {
   V( srv ); /* Wakeup server */
   busy_wait(); /* and let it run */
 }

 /* Wait for reply */
 while( !dequeue(Q[clnt],ans) ) {
   busy_wait(); /* Try to handoff */
   Q[clnt]->awake = 0;
   if( !dequeue(Q[clnt],ans) ) {
     P( clnt ); /* Wait for server */
     Q[clnt]->awake = 1;
   }
   else { /* reply ready */
     if( tas(&(Q[clnt]->awake)) )
       P( clnt ); /* Fix race condition */
     break;
   }
 } /* end while */
}
```

```
void Receive( Msg *msg ) {
 if( dequeue( Q[srv], msg ) ) return;

 yield(); /* Let clients run */

 while( !dequeue( Q[srv], msg ) ) {
   Q[srv]->awake = 0;
   if( !dequeue( Q[srv], msg ) ) {
     P( srv ); /* Wait for client */
     Q[srv]->awake = 1;
   }
   else { /* not empty */
     if( tas( &(Q[srv]->awake) ) ) P(srv);
     break;
   }
 } /* end while */
}

void Reply( int clnt, Msg *msg ) {
 while( !enqueue(Q[clnt],msg) )
   sleep( 1 ); /* queue full */

 if( !tas( &(Q[clnt]->awake) ) ) V(clnt);
}
```

Figure 7: **Both Sides Wait and Yield**: `busy_wait/yield` calls are added to the `BSW` algorithm to effect hand-off scheduling.

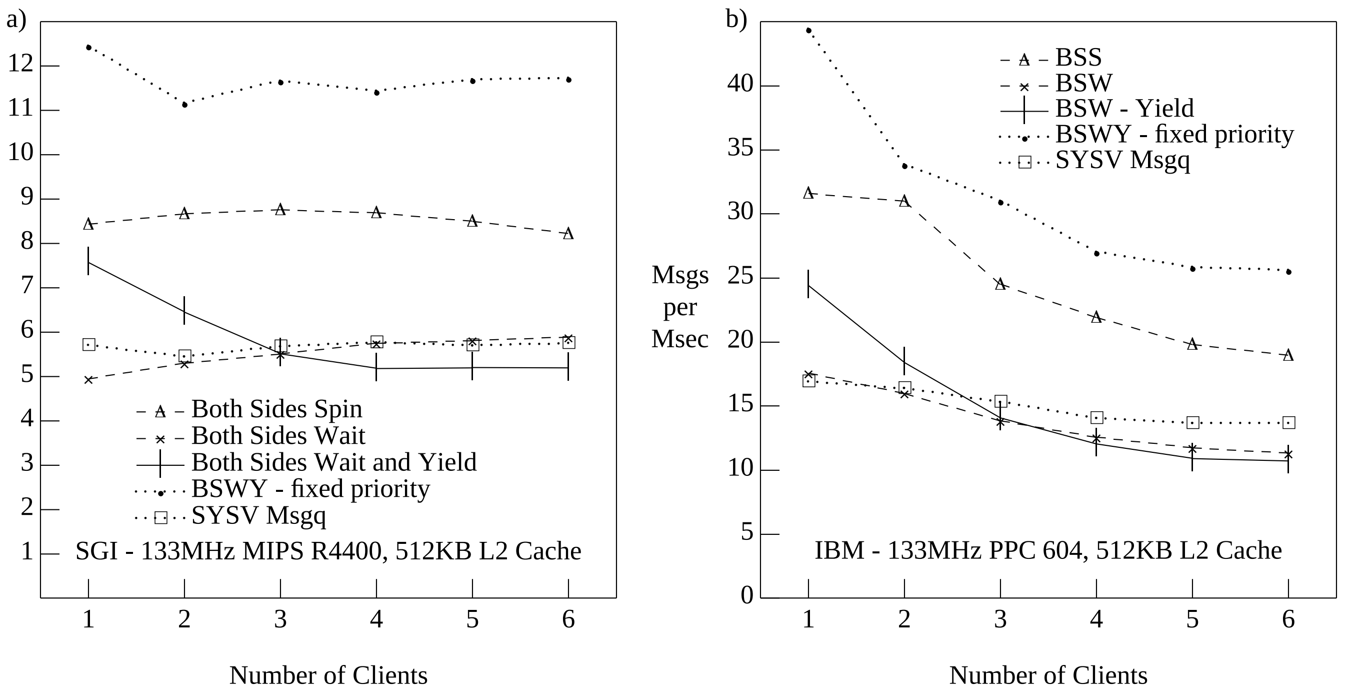

Figure 8: Measured uniprocessor Server Throughput in Messages/millisecond for varying numbers of client processes. The solid curve in each figure represents the throughput for the **Both Sides Wait and Yield** algorithm of Figure 7.

attempts to give the server one last chance to prepare a reply before the client puts itself to sleep.

Figure 7 also shows a `yield()` system call added in the Receive implementation of the server to allow the clients to process their replies and (possibly) enqueue their next request. Note that the server only yields if a dequeue attempt results in a queue empty condition. The reason for this extra dequeue is again for scalability with multiple clients. In particular, with multiple clients there is a higher probability that the server's queue has multiple outstanding entries, and it is more productive to continue processing than to give up the processor after every reply.

Figure 8 shows how the hand-off suggestions of the **BSWY** algorithm affect performance on a uniprocessor. The solid curves in the figure show the server throughput when the default scheduling mechanisms are used. The curves show that the `busy_wait` calls are effective for one or two clients, but that the performance degrades as concurrency is increased further. The reason is that the `yield` contains no hint about which process should be favored, and so *any* ready process, including the yielding process, could be scheduled as a result. Unless control is actually transferred to the server, the client will still block, in which case the `yield` calls have no benefit and actually contribute to the critical path latency.

The top dotted curves of Figure 8 show the performance of **BSWY** when fixed-priority scheduling is used. The throughput with this scheduling policy basically matches the performance of the busy-waiting **BSS** algorithm under the same scheduling policy (see Figure 3). While fixed-priority scheduling is not generally applicable to most client/server environments, we believe this performance could be obtained if hand-off scheduling were implemented.

```
void Send( Msg *msg, Msg *ans ) {
  while( !enqueue( Q[srv],msg ) )
    sleep( 1 ); /* queue full */

  if( !tas(&(Q[srv]->awake)) ) V(srv);

  spincnt = 0;
  while(empty(Q[clnt]) && spincnt++<MAX_SPIN)
    poll_queue( Q[clnt] );  /* Try to handoff */

  while( !dequeue(Q[clnt],ans) ) {
    busy_wait(); /* Try to handoff */
    Q[clnt]->awake = 0;
    if( !dequeue(Q[clnt],ans) ) {
      P( clnt ); /* Wait for server */
      Q[clnt]->awake = 1;
    }
    else { /* reply ready */
      if( tas(&(Q[clnt]->awake)) )
        P( clnt ); /* Fix race condition */
      break;
    }
  } /* end while */
}
```

```
void Receive( Msg *msg ) {
  spincnt = 0;
  while( empty(Q[srv]) && spincnt++<MAX_SPIN )
    poll_queue( Q[srv] ); /* Try to handoff */

  while( !dequeue( Q[srv], msg ) ) {
    Q[srv]->awake = 0;
    if( !dequeue( Q[srv], msg ) ) {
      P( srv ); /* Wait for client */
      Q[srv]->awake = 1;
    }
    else { /* not empty */
      if( tas( &(Q[srv]->awake) ) ) P(srv);
      break;
    }
  } /* end while */
}

void Reply( int clnt, Msg *msg ) {
  while( !enqueue(Q[clnt],msg) )
    sleep( 1 ); /* queue full */

  if( !tas( &(Q[clnt]->awake) ) ) V(clnt);
}
```

Figure 9: Both Sides Limited Spin (**BSLS**)

4.2 Adding Limited Busy-Waiting

The **BSWY** algorithm of Figure 7 shows some promise in that it seems to be able to effect a hand-off to the consumer for small numbers of clients. However, Figure 8 shows that real hand-off scheduling, as approximated by the fixed priority scheduler, could do much better. Remember that in Section 2.2 we found it took 2.5 `yield` calls (on average) before the default scheduler actually performed a context switch. It is therefore reasonable to conclude that the **BSWY** algorithm is unable to reach its full potential because the single hand-off suggestions (`yields`) inserted in the algorithm do not necessarily result in the desired context switch.

The **Both Sides Limited Spin (BSLS)** algorithm of Figure 9 incorporates a loop around the hand-off attempts. Note that both the server and client spin, the rationale is that the client is anxiously awaiting a reply; and it is cheaper to let the server spin a little than to wake it up if it goes to sleep. Note also that both the client and server can still block, which happens if there is no activity after `MAX_SPIN` dequeue attempts. The BSLS algorithm also makes use of an `empty` test that checks the head of the queue without locking it. The `poll_queue` subroutine is implemented as `yield` on a uniprocessor, and as a delay loop that calls `empty` on a multiprocessor.

Picking the right value for `MAX_SPIN` is important: it should be large enough so that the client will not block for the (common) case where the server has an immediate response; and it should be large enough to ensure that a hand-off will be effected. On the other hand, too large a value of `MAX_SPIN` will waste resources if the server has had to do I/O or is already busy handling other local requests. In either case the client should get out of the way and allow other useful work to be

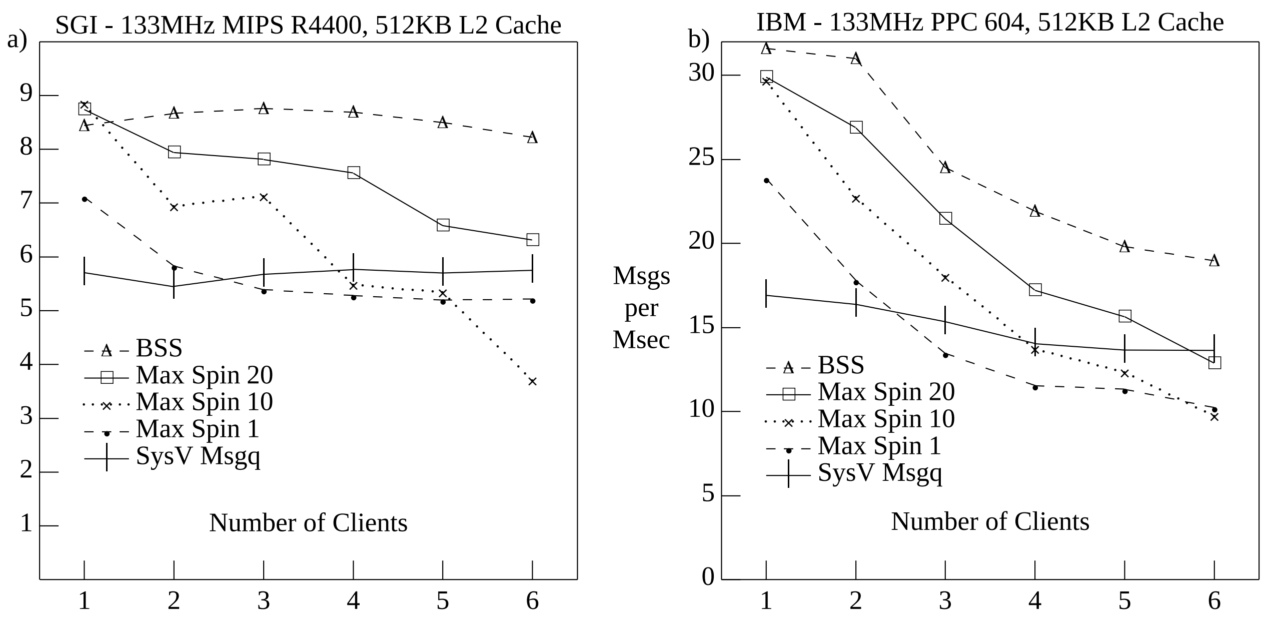

Figure 10: Measured uniprocessor Server Throughput in Messages/millisecond for varying numbers of client processes. The curves show the effect of different `MAX_SPIN` values for the **Both Sides Limited Spin** algorithm.

accomplished.

Figure 10 shows the sensitivity of the algorithm to `MAX_SPIN`. As expected, performance generally improves as the number of tries is increased. We verified that this is because the probability of falling out of the loop decreases as the number of hand-off attempts increases. At a `MAX_SPIN` value of 20, a single client only blocks 3% of the time, and gets an answer back within 2 iterations on average. Even with six clients, the numbers rise to: 10% of the loops fall-through; and 4 iterations of the loop are executed on average.

5 Multiprocessor Performance

Figure 11 shows the performance of the various algorithms on an 8-processor SGI Challenge. The code used in this experiment is identical to that used for the uniprocessor experiments, except that the `poll_queue` implementation uses a busy wait loop (25 μsec) where the `empty` check is made on every iteration.

The figure shows five curves: the top curve is the Both Sides Spin algorithm, the middle three curves are the Both Sides Limited Spin algorithm with different values of `MAX_SPIN`, and the bottom curve is for System V Message Queues. We see that System V Message Queues perform the worst and are unable to scale with increased concurrency demands. The best performance is for the BSS algorithm, whose throughput increases rapidly until the server saturates, and then stays stable. The Both Sides Limited Spin algorithms have similar performance to BSS up to a point, and then performance degrades rapidly. The reason for this is that as soon as a client spins for longer than `MAX_SPIN`, the server must pay the extra overhead to wake it up, which increases the load on the server and as a result increases the probability of other clients spinning for longer than

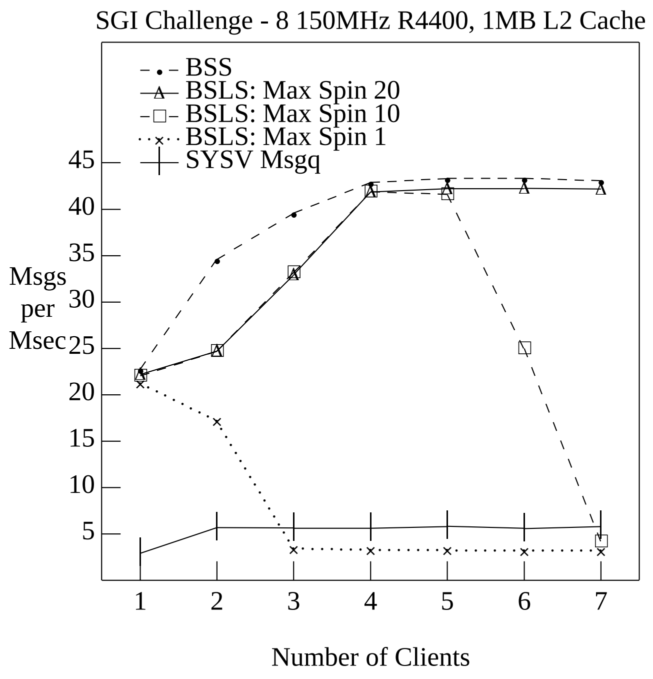

Figure 11: Measured multiprocessor Server Throughput in Messages/millisecond for varying numbers of client processes.

MAX_SPIN.

We could break the positive feedback in the BSLS algorithm by having the server recognize the fact that it is overloaded, and limit the number of clients it wakes up at any given time. The challenge is constraining the concurrency in this fashion while guaranteeing that starvation doesn't occur. We leave this for future work.

6 Modifying `yield` in Linux

Earlier sections have shown that fixed-priority scheduling substantially increases performance when the algorithms incorporate `yield` calls to suggest hand-off scheduling to the operating system scheduler. We would like to prove that a hand-off scheduling policy would indeed achieve the same performance as a fixed-priority policy, but without the deadlock and priority inversion problems that forces users of real-time scheduling to have super-user privileges.

Fundamentally, an operating system scheduler views multiple ready processes as *competing* for CPU resources. For this workload, the dynamically degrading priorities of typical schedulers will maximize throughput while still preventing starvation. However, since user-level IPC processes are *co-operating* instead of competing, overall system throughput would be best served with a hand-off scheduling policy. In the remainder of this section, we suggest an interface and implementation for a hand-off scheduling policy.

Ideally, a `handoff` system call would take a single `pid_t` argument (say, `pid`) that would be interpreted as follows:

`pid = some pid` - hand-off to the specified `some pid`. Of course, `some pid` must already be in the ready queue to be eligible for execution.

`pid = PID_SELF` - same semantics as `yield`.

`pid = PID_ANY` - block the calling process and allow highest priority ready process to run, *even* if it has a lower priority than the caller.

The first argument type (`pid = some pid`) would be used by the clients to *hint* that the server should execute. Since hand-off scheduling is often not incorporated because malicious processes could use it to monopolize the processor, a reasonable implementation might be to degrade the dynamic priority of `some pid` so that it would be favored to run but could not hog the CPU. The third argument style (`pid = PID_ANY`) would be used by the server to allow the clients to run. The server could use the first form to specify the client to whom it just responded, but in general it is sufficient for the server to inform the operating system simply that it has no useful work to do.

We implemented this `handoff` system call in the Linux 1.0.32 Slackware Release. It turns out that this version of Linux has a relatively simplistic scheduler that does not exhibit all of the characteristics of the other commercial operating systems we looked at. In particular, the support for fixed-priority scheduling was too immature to run the busy-waiting algorithms at all. Even with the default scheduler, we found that the response time for the busy-wait algorithm (**BSS**) was on the order of 33 *milliseconds* instead of the 120 microseconds we were expecting. The problem appeared to be in the way the dynamic priority was aged, so to fix it, we changed the `sched_yield` call to expire the caller's quantum and force a context switch. This change brought the latency back to 120 μsec on a 66MHz 486 machine. Of course, this is exactly the way we would like the commercial unix schedulers to treat `yield`, and so the results we obtain from the busy-wait algorithm in Linux should correspond to the fixed-scheduling curves we obtained from AIX or IRIX.

Figure 12 shows the measured throughput on the Linux system with our modified `sched_yield` implementation. The curves show that the **BSWY** algorithm – the one *without* any client side spinning – performs as well as the busy-waiting **BSS** algorithm. Because the way we changed `sched_yield` brought us near maximal performance, we actually found that our implementation of `handoff` matched the **BSWY** performance, but did not improve it further.

7 Related Work

With the recent interest in micro-kernels and the general trend in moving traditional operating system functionality to the user level, we were surprised that more work did not exist in this area.

The issues of fast user-level IPC arise in user-level thread packages [10], but since these packages also include the thread scheduler, system calls are not needed to sleep and wake-up threads. Also, the race conditions that we encounter as a result of crossing the kernel boundary do not have to occur in a user-level thread package because the interface can allow a thread to atomically sleep and set a flag to inform others that it is doing so.

Fast IPC has also been the focus of microkernel architecture research, and solutions based on hand-off scheduling have been proposed in Bershad's LRPC [1], Gamsa's PPC [4], Spring

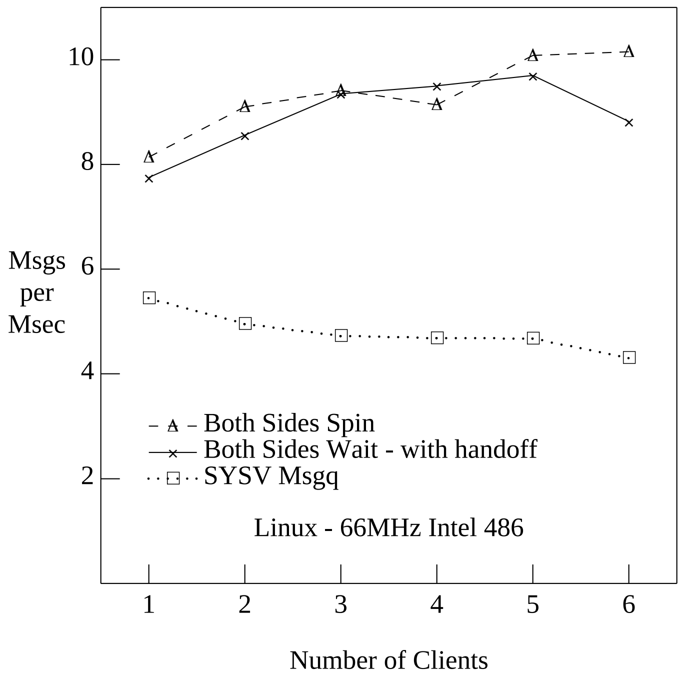

Figure 12: Measured uniprocessor Server Throughput in Messages/millisecond for varying numbers of client processes. The solid curve is the **BSWY** algorithm of Figure 7 with the modified `sched_yield` implementation.

doors [5], and Liedtke's L3 and L4 systems [7, 8]. However, all these systems are based on kernel mechanisms to support client upcalls and server registration of entry points.

Bershad's URPC system [2] is mostly implemented at the user level, but depends on a high degree of concurrency in the client to batch calls or tolerate latency when control must be handed to the server. In contrast, our work does not assume concurrency within a single client, and focuses on achieving low latency for the case where the server data is cached locally.

Many of the race conditions and scheduling considerations could be resolved with an extended operating system interface [6]. For example, SymUnix [3] allows user processes to indicate when they are executing a critical section, and should therefore not be pre-empted. We would welcome extensions such as our proposed `handoff( pid_t p )`, which could allow a client to suggest that the server be scheduled after a request has been enqueued.

8 Conclusion

The performance of IPC is crucial to many applications, and there has been a great deal of work done on developing IPC facilities for new operating systems that perform well. In this paper we have shown how, using user-level IPC, good performance can be achieved on current commercial operating systems. We showed that we could achieve throughputs on uniprocessors that are twice as high as the kernel mediated IPC facilities, and demonstrated that even better performance improvements can arise in multiprocessors. Also, we show that with just minor changes to the operating system interface, even better performance is possible.

Our work has so far mainly concentrated on achieving good performance for synchronous IPC on a uniprocessor, since this is the most challenging situation. In the future we intend to explore in more detail the other benefits of user-level IPC, namely, asynchronous IPC, specialized protocols, and multiprocessor performance. Also, more work is required on the security issues that arise when user-level IPC is used.

While the results in this paper are based on unrealistic micro-benchmarks, the motivation for this work comes from the performance and functionality limitations we found in current IPC facilities when developing a new data base server. This data base server exploits not only the improved performance of user-level IPC for synchronous IPC, but also the optimizations available for asynchronous IPC and customized protocols.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. "Lightweight Remote Procedure Call". *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[2] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. "User-Level Interprocess Communication for Shared Memory Multiprocessors". *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[3] J. Edler, J. Lipkis, and E. Schonberg. "Process Management for Highly Parallel UNIX Systems". In *Proc. USENIX Workshop on Unix and SuperComputers*, Pittsburgh, PA, September 1988.

[4] Benjamin Gamsa, Orran Krieger, and Michael Stumm. "Optimizing IPC Performance for Shared-Memory Multiprocessors". In *Proc. International Conference on Parallel Processing (ICPP)*, volume 2, pages 208–211, August 1994.

[5] Graham Hamilton and Panos Kougiouris. "The Spring nucleus: A microkernel for objects". In *Proc. Summer USENIX*, pages 147–159, Cincinatti, OH, June 1993.

[6] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler Conscious Synchronization. Technical Report TR 550, Dept. Computer Science, University of Rochester, Rochester, NY, December 1994.

[7] Jochen Liedtke. "Improving IPC by Kernel Design". In *Proc. 14th ACM Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, December 1993.

[8] Jochen Liedtke. "On μ-Kernel Construction". In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, December 1995.

[9] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, 1996.

[10] D. Stuart Ritchie and Gerald W. Neufeld. "User Level IPC and Device Management in the Raven Kernel". In *Proc. USENIX Microkernels and Other Kernel Architectures Symposium*, San Diego, CA, September 20–23 1993.