Regular Paper

Expedite: An Operating System Extension to Support Low-latency Communication in Non-dedicated Clusters

Hideo Saito,† Kenjiro Taura† and Takashi Chikayama†

We propose and evaluate Expedite, a simple operating system extension that allows parallel applications to communicate with low latency in non-dedicated clusters. We extend the UNIX API to allow a process to set the `O_EXPEDITE` flag of a socket, and revise the Linux scheduler so that a process is given a large priority boost when data arrives on a socket with the `O_EXPEDITE` flag set. Fairness is maintained by giving the boost only to processes that block most of the time, and by making the priority boost decay quickly. We describe an MPI implementation that takes advantage of this facility. Experiments show that the latency of collective operations such as `MPI_Scatter()` can be reduced with our facility, even when many high-priority processes are running in the background.

1. Introduction

1.1 Motivation

Clusters of workstations are rapidly replacing specialized supercomputing platforms. Over 40 percent of the machines on the TOP500 List [5] are clusters—three years ago, they represented just 10 percent.

Such clusters are often non-dedicated—while one user runs a parallel application, other users may be writing or compiling code. In the future, parallel applications may be running side by side with word processors and databases. In this kind of environment, parallel applications must often wait to be given CPU time. This wait—called a scheduling delay—lengthens communication latency and lowers the performance of parallel computation.

Yet scheduling delays can be avoided. It is the lack of support by operating systems that causes parallel applications to suffer these delays—commodity operating systems were not designed to support parallel applications with other applications running in the background. Our motivation is to give common operating systems functionality to support parallel applications in non-dedicated clusters.

1.2 Background

A parallel application can receive messages by either polling or blocking. Continuously polling is the fastest way to recognize incoming messages. However, there are environments and applications in which polling is not desirable. For example, we can improve the performance of an application by overlapping computation and communication. In a non-dedicated cluster, we can also improve the performance of the cluster as a whole by letting processes of other users run while waiting for messages.

Instead of polling, parallel applications may use a separate, blocking thread to handle messages. This thread would spend most of its time waiting for messages, but once a message arrives, it must be dispatched as quickly as possible—parallel applications require low latency for high performance, and scheduling delays increase latency.

Current operating systems, such as Linux, give high priority to processes that block often. As a result, blocking message handlers typically have high priority upon the arrival of a message. However, operating systems give equally high priorities to other processes that frequently perform blocking I/O or just sleep often. Thus, in a non-dedicated cluster, a message handler must compete with many other processes to be dispatched. When another process wins the competition and is dispatched first, the message handler must wait, increasing message latency.

For example, `top` and the blocking message handler of a parallel application are both given high priorities by operating systems. Thus, if `top` wakes up at the same time that a message arrives, operating systems are as likely to dispatch `top` as they are to dispatch the message handler. However, it would be much more desirable to let the message handler run first—`top` would not be noticeably affected by a few hundred microseconds' delay, but the parallel application would be.

† Department of Information and Communication Engineering, Faculty of Engineering, The University of Tokyo

In this paper, we propose Expedite, a simple operating system extension to allow parallel applications to communicate with low latency in non-dedicated clusters. We implement Expedite by making simple changes to the UNIX API and the Linux kernel. We extend the system call `fcntl()` so that it can set the descriptor flag `O_EXPEDITE`. We revise the Linux scheduler so that a process is temporarily given a large priority boost when it receives data on a socket with the `O_EXPEDITE` flag set. We preserve fairness by applying the priority boosts only to processes that consume little CPU time. In effect, Expedite allows applications to have a separate, "responsive" thread that is normally sleeping, but given high priority when a message arrives.

The rest of this paper is organized as follows. In Section 2, we discuss related work. We present the Expedite facility in Section 3. We discuss some simple experiments using the ping-pong test in Section 4, and then discuss a case study using MPI in Section 5. Finally, we conclude in Section 6.

2. Related Work

Today's popular operating systems use scheduling heuristics to dispatch interactive processes, such as editors and word processors, shortly after input. The basic mechanism is to maintain high priorities for processes that block often as employed by Linux [2], or to give a priority boost upon unblocking as employed by Windows. These schemes work fine for parallel applications as well, as long as they compete only with compute-mostly background load. Problems remain when a parallel application competes with processes that block often, such as interactive jobs, I/O-centric jobs, network daemons, and monitoring tools. Our proposal is a mechanism with which an application can ask the operating system to give a special kind of priority boost. The special boost is much larger than normal ones but decays much more quickly.

If only the needs of the message handler had to be addressed, a real-time operating system could be used. On commodity operating systems, real-time scheduling policies, which can raise the priorities of certain processes, could be used. Either solution would guarantee that the message handler is dispatched as soon as a message arrives. Yet such a guarantee gives certain processes an unfair advantage—this is not desirable in a shared environment, where the needs of many processes of many users must be addressed.

Schedulers with short dispatch delays have been studied extensively in the past. High resolution timers have been used in some systems such as RT-Mach [6]. However, the frequent interrupts caused by such timers produce an overhead that affects the performance of throughput-oriented applications. The overhead can be lowered by using soft timers [1], but this approach requires an overhaul of the existing timing mechanism of operating systems. Moreover, a high resolution timer mechanism is not useful to user-level applications without a responsive kernel, so a redesign of the operating systems themselves is necessary with this approach.

Research has also been performed to support time-sensitive applications on commodity operating systems. For example, proportion-based scheduling can be used to provide time-sensitive tasks low-latency scheduling while preventing them from consuming too much execution time. However, standard proportion-period scheduling requires an estimation of the proportion and period requirements of a process in advance, which is not realistic in a general-purpose environment. Time-Sensitive Linux [3] tries to address this by automatically assigning allocations to such tasks, but this approach still requires application-specific progress rate metrics, such as the fill-level of a bounded buffer of a producer or consumer.

3. Design and Implementation of Expedite

3.1 Overview

We design Expedite so that a parallel application can have a separate, "responsive" that is normally sleeping, but dispatched immediately after message arrival. A message-handling thread uses Expedite by setting the `O_EXPEDITE` flag of a socket and blocking until a message arrives. When a message arrives on the socket, Expedite gives the thread a large priority boost, preventing other processes from being dispatched first.

Expedite preserves fairness by only giving priority boosts to processes that consume little CPU time. Thus, a thread that blocks while waiting for messages benefits from Expedite, while a process that continuously polls for messages does not.

We implement Expedite with just a few changes to the Linux operating system. It does not require a redesign of the kernel or an overhaul of the existing timing mechanism as some attempts to support real-time systems do—it fits right into the existing code, minimizing the effects that it has on other parts of the operating system.

The simple design of Expedite also makes it possible to implement Expedite on many other operating systems besides Linux. `Fcntl()`, the interface that we use for Expedite, is available on other UNIX systems and even on some non-UNIX operating systems. The implementation we describe is specific to Linux, but similar modifications can be made to other operating systems that use priority-based schedulers and use thread libraries with a 1:1 scheduling model. Thus, non-dedicated clusters running a variety of operating systems can use Expedite to support parallel computation.

3.2 The Linux Scheduler

The scheduling cycle in Linux is called an epoch. At the beginning of an epoch, the scheduler allocates a time quantum to each process. For a process with the default priority, the time quantum is 6 ticks of the interrupt timer.☆ This value is decremented each time the process is running at the time of a timer interrupt, which occurs once every 10 milliseconds. The epoch continues until all processes in the run queue, the list of processes ready to run, have exhausted their ticks; a new epoch begins at this time, and processes are once again allocated time quanta.

`Schedule()` is called at certain times to allocate the CPU to processes. The function is called when the current process has used up its time quantum and a new process must be dispatched. It is also called when the current process blocks. Furthermore, it is called on return from interrupts and exceptions, before resuming execution of a user process. `Schedule()` calculates a value called `goodness` for each process in the run queue and dispatches the process whose goodness is the largest. In the default scheduling policy, `SCHED_OTHER`, goodness is calculated as follows:

```
goodness = counter - nice;
```

Here, `counter` is the number of ticks that the process has left in this epoch, and `nice` is a value that the user can set to statically raise or lower the priority.

Processes that sleep or block while waiting for I/O are removed from the run queue so that `schedule()` does not select them. Such processes have ticks left over when an epoch ends. Half of these ticks are carried over to the next epoch, raising their priorities. `Counter` of a process that continues to block or run only occasionally will converge to 11.

A non-dedicated cluster potentially has many processes that perform I/O or sleep. The Linux scheduler does not distinguish among them, giving them equally high priorities upon waking up. Thus, the scheduler may dispatch `top` or `emacs`, which can tolerate delays of millisecond order, before it dispatches the message handler of a parallel application, which can only tolerate delays of microsecond order.

3.3 Implementation of Expedite

The Expedite facility allows a process to declare if it wants to be given a priority boost upon the completion of certain I/O. This is accomplished by setting the `O_EXPEDITE` flag of a socket descriptor using the system call `fcntl()`:

```
fcntl(fd, F_SETFL, O_EXPEDITE);
```

If `select()` or `read()` is called to perform I/O on a socket with the `O_EXPEDITE` flag set, the `expedite` flag in the process descriptor of the calling process is set:

```
if (file->f_flags & O_EXPEDITE)
current->expedite = 1;
```

When the scheduler encounters a process that runs only occasionally and has its expedite flag set, the scheduler gives that process a large priority boost—enough to guarantee it higher priority than all other processes. We can identify a process that runs only occasionally, because such a process accumulates ticks. We give the boost to processes that have 10 or 11 ticks:

```
if (p->expedite && p->counter >= 10)
goodness += 100;
```

The process is given the priority boost until it is successfully dispatched. Once the process has been dispatched, normal scheduling resumes—the process runs with normal priority until it calls `select()` or `read()` again.

☆ For the i386 architecture.

By setting the `O_EXPEDITE` flag of certain socket descriptors, we let the operating system know when we need to be given a short time slice with little delay. This allows the operating system to give the message handler of a parallel application just enough time to respond to a message that has just arrived.

3.4 Fairness

A process is not able to use Expedite to steal long periods of time from processes with higher priority. While Expedite may cause a process with lower priority using `O_EXPEDITE` to be scheduled over a process with higher priority, Expedite only gives short time slices to processes that it prioritizes—until the next time that `schedule()` is called. As `schedule()` is called upon every timer interrupt, the time slice given by Expedite is at most 10 milliseconds. If an exception or interrupt causes `schedule()` to be called earlier, the time slice will be shorter.

We also prevent a process from gaining frequent priority boosts by using Expedite and blocking and unblocking often. A process that runs often loses ticks quickly, so such a process will not have the 10 ticks necessary to be given a priority boost. This is not a problem for message handlers, because they have an excess of ticks—they are blocking most of the time, and when they do run, they only run for a short while.

3.5 Applications of Expedite

The type of operations that benefit most from Expedite is short operations performed in response to requests from a remote node. Such operations can run independently from the "main" thread of an application. An example is the collective operations of MPI, such as `MPI_Scatter` or `MPI_Bcast`, which forward messages along a tree. By having a separate thread for collective operations, forwarding can be carried out independently from the main thread. With Expedite, such forwarding is guaranteed to be scheduled quickly even with high-priority background load. Another, perhaps more important application will be the remote memory operations of MPI-2 (get, put, and accumulate). Besides MPI, there are many frameworks that benefit from having message handlers that are predictably scheduled quickly, such as dynamic load distribution, distributed shared memory, and data migration.

4. Experiment Using the Ping-Pong Benchmark

In this section, we evaluate the basic concepts of Expedite by performing the ping-pong test under various conditions. As the ping-pong process spends most of its time blocking, it has high priority upon message arrival. We show, however, that with the existing Linux kernel, background processes that also have priority can cause the ping-pong process' being dispatched to be delayed and lengthen the round-trip time. We are able to reduce this delay by using Expedite and giving the ping-pong process a large priority boost when it receives a message. Our experiments show that this technique works even when multiple processes use Expedite. We also show that Expedite does not break fairness greatly, because it only boosts the priority of a process that consumes little CPU time.

We use two 866 MHz Pentium III machines with 1 GB RAM for our experiments. Both machines run the Linux 2.4.25 kernel, and are connected to a 100-Mbps Ethernet network.

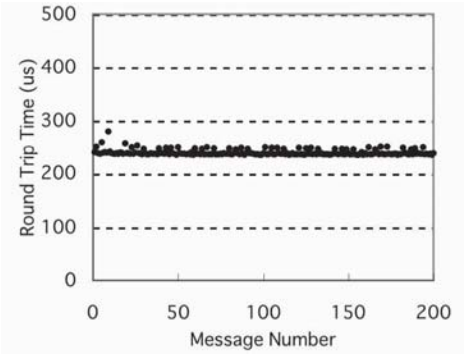
4.1 Effects of Background Processes

In the first experiment, we show how different background processes affect the round-trip time of ping-pong messages. We use 500-byte messages, and ping every 50 milliseconds. In one trial, we run in the background `compute-only`, a process that continuously computes the Fibonacci series. In a second trial, we add to the background two copies of `sleep-mostly`, a process that repeatedly computes for 5 milliseconds then sleeps for 45 milliseconds.

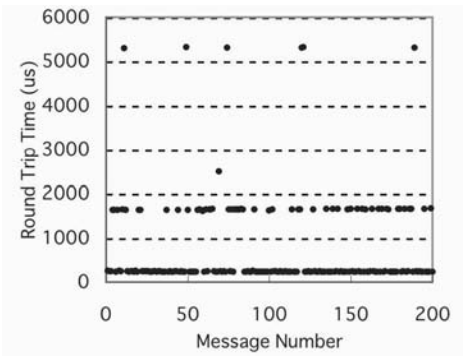
Figure 1 shows the results. `Compute-only`, despite using 99% of the CPU, caused no scheduling delays. However, when `sleep-mostly` was added, the ping-pong process occasionally suffered delays of millisecond-order. The ping-pong process had high priority, but so did `sleep-mostly`. When the scheduler dispatched `sleep-mostly` first, the ping-pong process had to wait.

4.2 Reducing Scheduling Delays with Expedite

In the second experiment, we show that by using Expedite, we can reduce the scheduling delays caused by a high-priority process running in the background. We perform the ping-pong test with `compute-only` and `sleep-mostly` running in the background again, but this time we have the ping-pong pro-

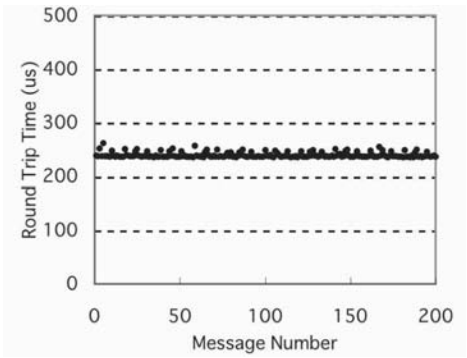


(a) `compute-only in BG`

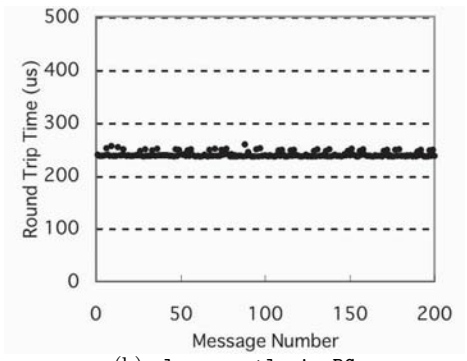


(b) `sleep-mostly in BG`

Fig. 1 Round-trip time of ping-pong messages with the existing Linux kernel.



(a) `compute-only in BG`



(b) `sleep-mostly in BG`

Fig. 2 Round-trip time of ping-pong messages with Expedite.

cess block on sockets with the `O_EXPEDITE` flag set. As the ping-pong process is normally blocking, it accumulates ticks. Thus, when a "ping" or a "pong" arrives, it has the 10 ticks necessary to be given a priority boost by the scheduler. This prevents `sleep-mostly` from being scheduled before the ping-pong process.

Figure 2 shows the results. This time, `sleep-mostly` did not cause any significant delays.

4.3 Effects of Multiple Processes Using Expedite

In the next experiment, we show that multiple processes can use Expedite to reduce scheduling delays without a large side-effect on other processes. We have ten pairs of processes perform the ping-pong test on two nodes, and run `compute-only` and `sleep-mostly` in the background. In the first trial, none of the processes uses Expedite. In the second trial, nine of the pairs use Expedite, and one does not.

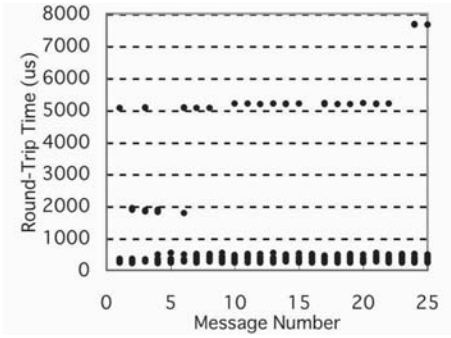
Figure 3 shows the results. In the first trial, all pairs suffer scheduling delays. In the second trial, the nine pairs that used Expedite stopped suffering delays, but the last pair that did not use Expedite continued to suffer delays.

While Expedite simultaneously helped nine pairs of processes perform low-latency communication, it did not affect the performance of other processes much. The pair of processes that did not use Expedite suffered slight delays (**Fig. 4**), but much smaller ones compared to those caused by `sleep-mostly`. The throughput of `compute-only` and `sleep-mostly` (measured by the number of Fibonacci numbers computed per second) decreased by less than one percent.

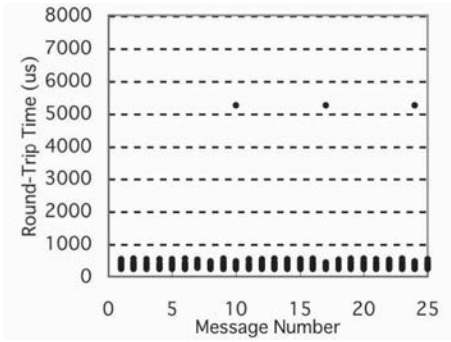
4.4 Fairness

The last experiment shows that a process that sets the `O_EXPEDITE` flag and consumes a lot of execution time does not break fairness. We run `compute-only` and two copies of `sleep-mostly` in the background, and have the ping-pong process consume CPU time by performing some extra computation—it computes the 16th element of the Fibonacci series 18 times after sending each "pong." This takes 1.6 milliseconds.

Figure 5 shows the results. Scheduling delays occurred even though Expedite was used—

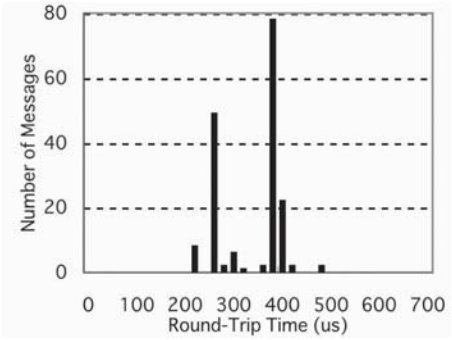


(a) `No pairs using Expedite`

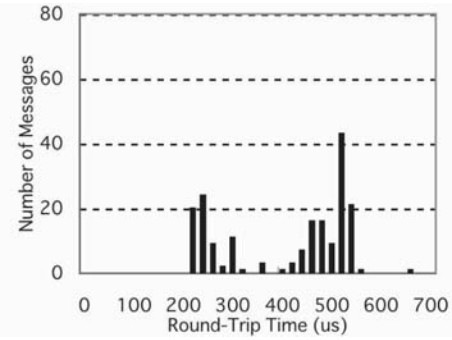


(b) `9 pairs using Expedite`

Fig. 3 Round-trip time when 10 pairs of processes simultaneously performed the ping-pong test.



(a) `Other pairs not using Expedite`



(b) `9 other pairs using Expedite`

Fig. 4 Round-trip times for the pair that did not use Expedite.

and they occurred just as many times as when Expedite was not used. Expedite did not help the ping-pong process, because the extra computation made it impossible for the process to maintain 10 ticks.

5. Low-Latency Collective Operations with Expedite

In this section, we discuss a case study using MPI to show how Expedite can improve the performance of parallel computation in a non-dedicated cluster. We begin by showing that `top`—a process commonly used by cluster users—running in the background can have a large effect on a collective operation such as `MPI_Scatter()`, especially in polling implementations such as MPICH [4]. We then show that a blocking implementation performs better, but is still affected by background processes. Finally, we show that we can improve the performance of the blocking implementation by using Expedite.

5.1 Polling Implementation of MPI

Many implementations of MPI poll for messages in order to avoid the overhead of context switching that comes with blocking. In

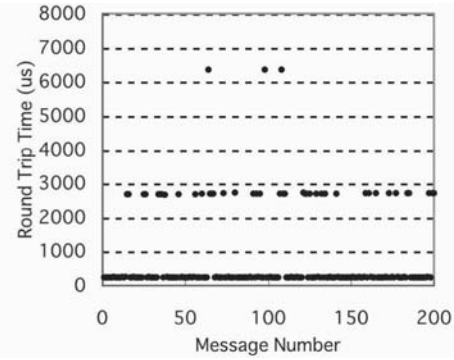


Fig. 5 Round-trip time when the ping-pong process performed 1.6 milliseconds of computation after sending each "pong."

MPICH, for example, messages are polled for whenever an MPI call such as `MPI_Send()` or `MPI_Recv()` is made. The process blocks only if the desired message is not found at the time of polling.

However, for collective operations such as `MPI_Scatter()`, not polling until an MPI call is made can delay the entire operation. In MPICH, `MPI_Scatter()` is performed by forwarding messages along a tree (**Fig. 6**). Thus,

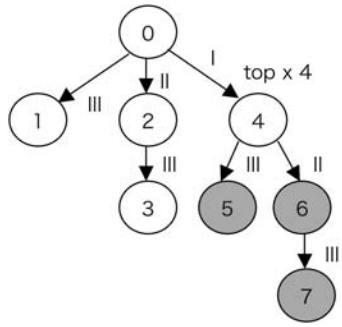
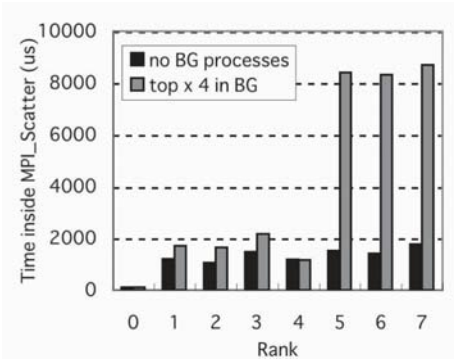
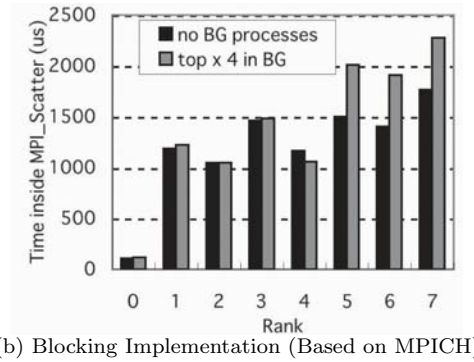


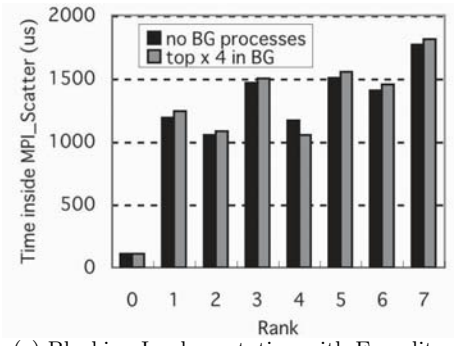
Fig. 6 The tree used in an eight-node `MPI_Scatter()`.



(a) Polling Implementation (MPICH)



(b) Blocking Implementation (Based on MPICH)



(c) Blocking Implementation with Expedite

Fig. 7 The time that each node spent on `MPI_Scatter()` with `top` running on rank 4. Average of 35,000 trials.

if a non-leaf node delays calling `MPI_Scatter()`, all its children are delayed as well. In an eight-node `MPI_Scatter()`, for example, a delay in rank 4 would propagate down the tree and also cause delays in ranks 5, 6, and 7.

We perform an experiment to show how the delay caused on one node during an `MPI_Scatter()` can propagate to other nodes and cause delays there as well. We perform an eight-node `MPI_Scatter()` in which each node performs some computation before calling `MPI_Scatter()`. We use mpich-1.2.5.2 as our MPI library, and on rank 4, we run 4 copies of `top` updating with a one-second delay (`top -d 1`) in the background. The amount of computation that each node performs before calling `MPI_Scatter()` is the same (120 milliseconds without background computation). However, rank 4 takes longer to reach `MPI_Scatter()`, because it must compete with `top`. We perform an `MPI_Barrier()` after the `MPI_Scatter()`, and repeat the experiment 35,000 times.

Figure 7 (a) shows the average time that each node spent inside `MPI_Scatter()`. Ranks 5, 6, and 7 waited for over six milliseconds until `MPI_Scatter()` was called on rank 4. Co-existing with `top`, the computation preceding `MPI_Scatter()` was delayed on rank 4. As the call to `MPI_Scatter()` could not finish on ranks 5 and 6 until rank 4 passed the message along, the delay in rank 4 also caused a delay in ranks 5 and 6. This in turn caused a delay in rank 7.

5.2 Collective Operations Using Blocking Wait

Next, in order to reduce the propagation of delays, we revise MPICH so that collective operations such as `MPI_Scatter()` are handled by a separate message-handling thread that blocks while it waits for messages. When the message-handling thread receives a message, it passes the message along to its children, regardless of whether the main thread has already made the call to `MPI_Scatter()`. Thus, even if a non-leaf node delays making the call to `MPI_Scatter()`, its children are not delayed.

Figure 7 (b) shows the average time that each node spent on the `MPI_Scatter()` operation when we repeat the previous experiment with the revised implementation. We compute the time spent on the operation as the sum of the time spent inside `MPI_Scatter()` and inside the message handler before the call to `MPI_Scatter()`. The wait that ranks 5, 6,

and 7 suffered was successfully reduced by over 5 milliseconds. On rank 4, the computation preceding `MPI_Scatter()` was still delayed by `top`. However, the message-handling thread received and passed along messages independent of whether the main thread had made the call to `MPI_Scatter()`. Thus, the scatter operation did not pause while waiting for rank 4 to call `MPI_Scatter()`, and ranks 5 and 6 were not made to wait as in the polling implementation.

However, ranks 5, 6, and 7 still spent 500 microseconds longer inside `MPI_Scatter()` than when there were no background processes. This is because when a message arrived to rank 4, the message-handling thread had to compete with background processes to be dispatched. As both `top` and the message-handling thread were blocking most of the time, both had high priorities upon waking up. Thus, if `top` had just woken up when a message arrived, the scheduler sometimes dispatched `top` first. This scheduling delay caused ranks 5 and 6 to wait inside `MPI_Scatter()`. This in turn caused rank 7 to wait.

5.3 Eliminating Scheduling Delays with Expedite

Finally, we further revise the blocking implementation in order to reduce unwanted scheduling delays: we have the message-handling thread wait for messages on a socket with the `O_EXPEDITE` flag set. As the message-handling thread only runs occasionally, we expect it to be given a priority boost upon message arrival.

Figure 7 (c) shows the results. The time spent on `MPI_Scatter` was almost the same as when there were no background processes. When a message arrived on rank 4, the message-handling thread had to compete with background processes to be dispatched. Using Expedite, however, the message-handling thread was given a priority boost. Thus, the message-handling thread won the competition, and was dispatched without a scheduling delay. As the message-handling thread on rank 4 was able to pass on the scatter message promptly, ranks 5, 6, and 7 were not made to wait unnecessarily.

6. Conclusions

In this paper, we proposed and evaluated Expedite, an operating system extension to allow applications to use a separate, blocking thread to communicate with low latency in a non-dedicated cluster.

Expedite was able to prevent background processes from delaying ping-pong messages. With the existing Linux kernel, a blocking ping-pong process suffered scheduling delays of over 5 milliseconds. By using Expedite, these delays were eliminated.

Expedite was also able to support collective operations in MPI. With MPICH, `top` running on one of the nodes delayed the scatter operation by over 6 milliseconds. By adding a separate thread to handle scatter messages and using Expedite, the delays were eliminated.

While Expedite supported parallel applications, it did so fairly. Expedite supported blocking message-handling threads while they consumed little CPU time, but it ceased to provide support when they consumed too much time.

As non-dedicated clusters become more and more popular, an operating system extension such as Expedite will be very effective in supporting low-latency communication.

References

1) Aron, M. and Druschel, P.: Soft timers: Efficient microsecond software timer support for network processing, *ACM Trans. Comput. Syst.* (2000).
2) Bovet, D. and Cesati, M.: *Understanding the Linux Kernel*, O'Reilly & Associates (2001).
3) Goel, A., Abeni, L., Krasic, C., Snow, J. and Walpole, J.: Supporting Time-Sensitive Applications on a Commodity OS, *5th Symposium on Operating System Design and Implementation* (2002).
4) http://www unix.mcs.anl.gov/mpi/mpich/: MPICH.
5) http://www.top500.org/: TOP500 Supercomputer sites.
6) Savage, S. and Tokuda, H.: RT-Mach timers: Exporting time to the user, *USENIX 3rd Mach Symposium* (2003).

(Received January 31, 2004)
(Accepted April 27, 2004)

Hideo Saito is an M.S. candidate in Information and Communication Engineering at The University of Tokyo. He was born in Boston, USA in 1981. He received his B.S. degree from The University of Tokyo in March, 2004. His major research interests are parallel/distributed systems and operating systems.

Kenjiro Taura is associate professor of Information and Communication Engineering at The University of Tokyo. He was born in 1969 and received his B.S., M.S., and Ph.D. degrees from The University of Tokyo in 1992, 1994, and 1997. His research interests include parallel/distributed computing and programming languages. He is a member of ACM and IEEE.

Takashi Chikayama is professor of Frontier Informatics at The University of Tokyo. He was born in Tokyo, Japan in 1953. He received his B.S., M.S., and Ph.D. degrees from The University of Tokyo in 1977, 1980, and 1982. He worked for the Institute for New Generation Computer Technology from 1982 until 1995, when he became associate professor of Electronic Engineering at The University of Tokyo. His recent research activities include concurrent and parallel programming systems, parallel processing applications, and a unified computation complexity model for distributed and parallel processing.