Latency Reduction using a Polling Scheduler

Jørgen S. Hansen Eric Jul

Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, 2100 Copenhagen, Denmark
cyller,eric@diku.dk

Abstract

For many cluster network technologies, a considerable part of the communication latency experienced by applications stems from interrupt handling. To reduce network communication latency, many scientific parallel applications and communication libraries for such applications continuously poll the network adapters when waiting for incoming communication. This approach is hard to use in a more general cluster computing setting, as such busy waiting wastes processor resources and is vulnerable to long delays caused by the execution of compute bound processes. We propose integrating polling with the operating system scheduler, as the scheduler has extensive knowledge of the states of the processes, and thus is in a good position to decide when to poll. We have implemented such a polling scheduler in the Linux operating system. Our results show a latency reduction by 56% for small messages when using a stream protocol for a Scalable Coherent Interface (SCI) cluster network.

1 Introduction

In current high performance cluster networks, the communication latency of small messages may be dominated by the overhead of interrupt handling. Furthermore, when nodes in a cluster are confronted with high network loads, interrupt handling may cause performance degradation due to application starvation caused by high priority interrupt handling. This paper concerns the problems related to providing efficient and stable network event handling for clusters of workstations and network servers. Our approach is to let the processor poll for incoming network events both on each scheduling decision and whenever the system is idle. Processor resources need not be wasted as the scheduler is able to schedule tasks with low overhead as soon as they become runnable. The two main complications of using polling are the overhead of polling the devices and keeping the system responsive. We use event flags in physical memory to reduce polling overhead and we selectively enable interrupts based on process priorities to increase system responsiveness.

The rest of this paper is organised as follows. Section 2 describes our polling approach in detail. Section 3 explains how the polling approach is used with a stream protocol implementation for Scalable Coherent Interface (SCI) cluster networks. Performance measurements of this implementation are presented in section 4. Related work is covered in Section 5, and our conclusions are drawn in Section 6.

2 Scheduler Supported Polling

In this section, we present scheduler supported polling and describes how it can be used to decrease network communication latency and increase the stability of systems processing large amounts of network related events. The integration of polling with the process scheduling depends on the scheduler used. In this section, we base our discussion on a priority-based scheduler using preemptive round robin scheduling with the same priority. We first describe the basic mechanism for integrating polling with an operating system scheduler, then we look at how responsiveness of the system can be maintained in the face of low priority, processor intensive applications through the use of interrupts, and finally we examine how the use of threads can improve network handling stability.

2.1 Integrating Polling with the Scheduler

We propose integrating polling with the kernel scheduler to allow the kernel to poll network adapters both when performing scheduling decisions and during periods of idleness. The actual polling of I/O adapters across an I/O bus can be a time consuming task, and should be avoided. We base our polling on event flags placed in physical memory. A network adapter needing attention must be able to raise its associated event flag to request kernel processing. On each scheduling decision the scheduler checks these flags to detect pending events. The cost of checking these flags in the case where no attention is required, will be very low, because the flags are likely to be located in the second level cache. When an event is detected, the scheduler disables further activations on the flag and either executes a handler or schedules a kernel thread associated with the event flag. When the interrupt handler or the kernel thread is done processing events from the adapter, they reenable the polling of the event flag.

A network server processing a stream of incoming packets should not be forced to relinquish the processor to allow the polling to take place, as could happen if new arrivals were only processed by the handler or kernel thread. Instead, the scheduler supported polling should be combined with lazy receiver processing [5], where a process itself handles as much of the processing of incoming packets as possible.

One possible problem with postponing the servicing of network adapters with on-board buffering is that buffer overflow may occur during long packet bursts. If polling-based network handling is used to decrease the number of context shifts for a server applications, it can be worthwhile to pay attention to the network adapters to avoid packet loss. Certain modern network adapters allow the generation of high water mark interrupts [1], and by selectively enabling these interrupts, buffer overflow can be avoided. If, on the other hand, polling is used to increase stability, buffer overflow is part of the price for the increased stability during high load situations.

2.2 Using Priorities to Increase Responsiveness

Simply having the scheduler poll the event flags on scheduling decisions and during idleness can seriously degrade responsiveness. The worst case delay from when a packet arrives until the operating system notices the event is bounded by the time slice used by the scheduler. Thus, if a low priority process coexists with a high priority network server process, and the low priority process often uses its whole time slice, the responsiveness of the system will suffer. Thus, it is clearly not a good idea to disable interrupts in all cases, e.g., if a low priority process is scheduled while a high priority network server is waiting for data supplied by a network adapter then the processing of the incoming data for the network server may be delayed considerably (in modern UNIX systems up to 100 milliseconds). To avoid this we associate a priority in the regular application priority range with each network adapter. We use this priority to selectively enable interrupts from a network adapter when a process with a lower priority than that of the network adapter is scheduled. During idleness interrupts are disabled for all adapters using the scheduler supported polling. This selective enabling of interrupts incurs the overhead of interrupts when low priority processes are executing, but at the same time ensures both responsiveness and fair allocation of resources to low priority processes.

Thus, in systems relying heavily on asynchronous network communication, e.g., a software distributed shared memory application where fetching a remote page halts the execution of part of the parallel program, the network adapter should be assigned a higher priority than the application. This will allow a busy node to react with the usual responsiveness to incoming requests, and still allow idle nodes to react even faster.

2.3 Increasing Stability using Polling Threads

Research by Mogul et al. [8] has shown that kernel threads can increase the stability of network handling. In their approach the kernel threads suspend themselves, when they detect starvation problems. Our use of priorities associated with a network adapter extends easily to cover the polling threads too. When using scheduler supported polling, the polling thread is activated when an event flag is raised, and the polling thread then repeatedly polls the interface until there are no more events. The thread then reactivates the event flag and puts itself to sleep. As the polling thread has a priority in the range of regular processes, fair allocation of processor time is ensured. Thus, if a network server has higher priority than the polling thread, the processing of already arrived packets have higher priority than the processing of incoming data.

3 Scheduler Supported Polling for SciStream

We have implemented the scheduler supported polling scheme described in the previous section as a modification to the scheduler of the Linux operating system version 2.0.35. The implementation includes an API allowing devices to allocate and deallocating event flags. When a device allocates an event flag, it must specify an event flag handler, and call-backs to enable and disable interrupts.

This implementation has been used to experiment with scheduler supported polling for SciStream—a TCP compatible stream protocol for Dolphin's SCI cluster adapters [4]. The basic data transfer mechanism used by SciStream is remote DMA, and asynchronous remote event notification is supported through remote interrupts. In this paper, we focus on the overhead of the event notification mechanisms used by SciStream—further information on SciStream can be found in [6].

We have implemented two versions of network event handling mechanisms for the SciStream protocol using the scheduler supported polling support in Linux: one version performs all event handling directly in the handler raised by an event flag and one version uses polling threads scheduled by the handler. The basic event handling in SciStream works as follows.

When the SciStream protocol is activated, it allocates two event flags—one for handling notification of incoming data and one for handling connection management. These event flags are exported to each remote node connecting to the local node. If the SciStream protocol is deactivated the flags are invalidated at the remote nodes, and subsequently deallocated. As the event flags do not reflect a single real device, we think of the state associated with an event flag as a pseudo device. These pseudo devices can be in three different states: (1) interrupts are enabled, (2) interrupts are disabled and the event flag is not raised, or (3) interrupts are disabled and the event flag is raised. The event flag is used to represent these three states such that a remote node is able to access and modify the state through the exported event flag. When interrupts are enabled the event flag is zero, when interrupts are disabled and the flag is not raised the event flag is one, and when interrupts are disabled and the event flag is raised the value of the event flag is greater than one.

When a remote node signals an event, it reads the remote event flag using a special mapping that will automatically increment the value of the event flag. This will either disable interrupts if interrupts were enabled or raise the event flag. If the remote node reads a zero value, it has disabled interrupts and must generate an interrupt. In either case the appropriate handler is activated.

4 Performance

This section presents performance measurements of the Linux implementation of scheduler supported polling. We examine two aspects of the scheduler supported polling: (1) the overhead of extending the operating system scheduler, and (2) the effects on SciStream packet processing overhead of scheduler supported polling. The measurements were carried out on workstations equipped with 450MHz Pentium II processors.

4.1 Event Handling Scheduler Overhead

The Linux scheduler has been extended with two pieces of code: the polling of the scheduler flags, and the enabling and disabling of interrupts when a process with priority lower than a device is scheduled. To measure the overhead added by these two pieces of code, we augmented the scheduler with measurements using the Pentium processor cycle counters. We implemented a kernel module (a pseudo-device) that registers event flags as a real device would. The action taken by the module, when an event flag is raised, is simply to find the event flag handle in an array and reenable the event flag (corresponding to a small loop and one function call). The enabling and disabling of interrupts simply increments a counter. We measured three configurations using from 0 to 16 allocated event flags with a size of four bytes each:

No Activity We let the pseudo-device allocate a number of event flags, but the event flags are never raised. This indicates the minimal overhead for scheduler supported polling, as the event flags are examined on each scheduling decision without activating any event flag handlers.

Remote Activity This configuration is the same as **No Activity** with the modification that an additional module has been loaded on a remote node. This module continuously raises the event flags allocated by the pseudo-device module, resulting in continuous activity on the event flags.

Enabling and disabling interrupts The last configuration includes two user-level processes, one with higher priority than the pseudo-device and one with lower priority. These two processes continuously make voluntary process yields. Thus, on each scheduling decision there will be a change in the interrupt status of the pseudo-device.

The overhead of the different configurations is shown in Figure 1. The overhead added in the case where no flags have been allocated, amounts to 67 nanoseconds, and each additional flag adds an overhead of 25 nanoseconds. In the case of remotely generated activity, the overhead is 135 nanoseconds for a single flag, and 91 nanoseconds are added for each flag. For the change in interrupt status, the overhead in the case with no allocated flags is 56 nanoseconds, and for a single allocated flag the overhead is 180 nanoseconds. Additional flags add 39 nanoseconds of overhead. This is a bit more expensive than we speculated in Section 2, but our current implementation might be optimised for better cache locality.

As an example of the overhead in a typical configuration, consider the case where a single event flag is raised in a system with a total of five allocated event flags. The overhead added by the scheduler for activating the event handling procedure associated with the flag would be $67 + 4 \times 25 + 91 = 258$ nanoseconds.

We measured the cost of a full context shift in Linux to be 4.2 microseconds. Thus, scheduler supported polling adds between 1.6% and 11% overhead to the process scheduling in the case of no activity. The scheduling of low priority processes may suffer further due to the cost of changing the interrupt status.

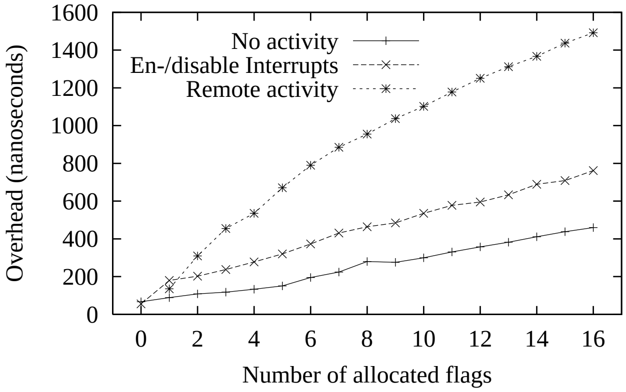

Figure 1: Overhead of Scheduler Supported Polling

4.2 Event Handling Packet Processing Overhead

The SciStream protocol supports four different kinds of event handling mechanisms: application polling, scheduler polling, scheduler polling with threads, and interrupts. In this section, we examine the effect of these different approaches on application to application latency. All measurements reported are the average latency of 10,000 request-response exchanges of one byte packets using simple benchmark applications. To ensure that a receiving process is actually blocked waiting for the reply, the sender executes a waiting loop before transmitting a packet. The duration of the waiting loop is approximately 80 microseconds.

We examine the latency of the following four basic configurations: (1) **AppPol** uses a SciStream read operation that continuously checks a single connection for incoming data, but does a processor yield between each check. This configuration does not support the select system call, (2) **SchedPol** uses scheduler supported polling without the polling threads, (3) **SchedPolThread** uses scheduler supported polling with polling threads, and (4) **Interrupt** uses a remote interrupt to notify a receiver of available data. For each of these configurations except **AppPol** we also have a ***XX*Hog** version that is the same as the ***XX*** version except for that one side is also running a processor hog application that only yields the processor when preempted. This shows the behaviour of scheduler supported polling on hosts running low priority compute-intensive jobs,

Table 1 shows the latency of a one byte packet for these configurations. The **AppPol** configurations results in the lowest latency with 89.69 microseconds for a one byte packet. The **SchedPol** configuration adds another 17.7 microseconds to this overhead. This added overhead is due to the remote event notification, and selecting the proper process for execution. **SchedPolThread** further increases the latency by 2.3 microseconds to 109.7 microseconds due to the extra context switch caused by the polling thread. The existence of the processor hog application increases the latency by another 32.6 to 37.2 microseconds (as the hog was only executing on one side of the test setup, the latency in Table 1 only show half the overhead). Using interrupts further increases the latency by 20.1 microseconds to a total of 129.8 microseconds. The **InterruptHog** configuration is unaffected by the presence of the hog.

Event Handling Overhead		
Configuration	Time in microseconds	
	1 byte packet latency	Added overhead (compared to AppPol)
AppPol	89.7	-
SchedPol	107.4	17.7
SchedPolHog	126.0	54.9
SchedPolThread	109.7	20.0
SchedPolThreadHog	126.0	52.6
Interrupt	129.8	39.9
InterruptHog	129.4	39.5

Table 1: Event Handling Overhead

Without the waiting loop the **AppPol** configuration results in a latency of 10.1 microseconds. Using the event notification of Linux, the improvement in latency of the scheduler supported polling (17.7+10.1 = 27.8) is reduced to 56% when compared to interrupts (39.9 + 10.1 = 50). The cost of using the SciStream implementation of scheduler supported polling in the presence of low priority processes is an increase in latency by 13.1 to 15.4 microseconds due to the two stage process of generating the remote interrupt. Future versions of the driver software for the SCI adapters will allow this to be performed in one operation, resulting in a latency equivalent to that of a regular interrupt.

5 Related Work

Polling Watchdogs [7] consider polling network adapters when scheduling decisions are made, but they do not consider the use of flags in physical memory as a means to reduce the polling penalty. To increase the responsiveness of the system, and to reduce the likelihood of buffer overflow in the network hardware, special purpose hardware is used to perform interrupts in case a scheduling decision, and hence a polling operation, is not performed within a reasonable time-frame.

Ensuring responsiveness when the network application programmers interfaces do not support interrupts has been investigated by Perkovic et al. [9]. They describe an approach to support low overhead network polling using event flags similar to ours, but they mainly consider inserting this polling code into applications through source-code manipulation.

A technique to provide efficient timer support is soft timers [2], where the operating system is sprinkled with timer checks at places, where a timer handler can be fired with low overhead. Soft timers can be used for efficient clocked polling, but the approach does not increase the responsiveness of an idle system.

In scientific parallel programs, the program structure, e.g., a computation followed by a data exchange, can result in processes being co-scheduled. Thus, it might be beneficial to poll for arriving data for a limited time before blocking. Damianakis et al. [3] experiment with using fixed busy-wait thresholds before blocking on receive operations when using the user-level communication libraries for the SHRIMP network. They are able to reduce communication overhead for a range of different workloads when compared to pure blocking. Processor cycles may still be wasted in clusters executing a mix of parallel and sequential processes. Schmitt et al. [10] address this problem by monitoring the load of a cluster as a whole, and adjusting the polling threshold according to the observed load.

The work on eliminating livelock [8] examines the behaviour of network servers during high network load. The problem of livelock, is explored in great detail. A solution is suggested, where the interrupt is only used to schedule a polling kernel thread—and letting this thread do the actual network processing. The approach differs from ours in that interrupts are used for scheduling the kernel threads and that the kernel threads have higher priority than any user process.

6 Conclusions and Future Work

By integrating polling with the operating system scheduler, we are able to provide low latency handling of events when the processor load is low. The overhead introduced by this kind of polling is very low, if events are signalled through flags placed in main memory. We integrate polling threads with the scheduling of normal processes by assigning the network adapters a priority in the range of the normal processes. This allows network handling to be prioritised higher than the execution of some applications, but lower than others. This concept of a priority of device handling is also applied to the actual polling—to keep the responsiveness offered by interrupts, interrupts for devices are enabled when, and only when, running processes with priority lower than the device priority.

The use of scheduler supported polling in a multiprocessor environment is under investigation, as is the integration of polling with other scheduling policies. Other future work includes providing applications with access to event flags.

Acknowledgements

Kåre Løchsen and Hugo Kohmann from Dolphin Interconnect Solutions (Norway) willingly granted us access to the source codes for the PCI-SCI adapter and answered all of our questions. Roger Butenuth from Universität Paderborn (Germany) kindly gave us access to his Linux port of the PCI-SCI driver. The implementers of SciOS, both Povl Koch from DIKU and Emmanuel Cecchet and Xavier Rousset de Pina from the SIRAC group at INRIA, provided us with valuable support for the SciOS prototype.

References

[1] AceNIC. See http://www.alteonwebsystems.com/ for a description of the AceNIC.

[2] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 232–246, Kiawah Island Resort, near Charleston, South Carolina, December 1999.

[3] S. Damianakis, Y. Chen, and E. Felten. Reducing waiting costs in user-level communication. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS-97)*, pages 381–387, Los Alamitos, April 1–5 1997. IEEE Computer Society Press.

[4] Dolphin Interconnect Solutions. PCI-SCI cluster adapter specification, May 1996. Version 1.2.

[5] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *The Second Symposium on Operating Systems Design and Implementation (OSDI '96) Proceedings*, pages 261–276, October 1996.

[6] Jørgen Sværke Hansen, Povl T. Koch, and Eric Jul. A stream protocol implementation for an SCI-based cluster of workstations. In *Proceedings of the 1999 Workshop on Cluster-Based Computing*, pages 16–20. ACM, June 1999. Held in conjunction with the 1999 International Conference on Supercomputing.

[7] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin Theobald, and Xinmin Tian. Polling watchdog : Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 179–190, New York, May 22–24 1996. ACM Press.

[8] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[9] Dejan Perkovic and Peter J. Keleher. Responsiveness without interrupts. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 101–108. ACM, June 1999.

[10] Michael Schmitt, Maximilian Ibel, Anurag Acharya, and Klaus E. Schauser. Adaptive receiver notification for non-dedicated workstation clusters. In *Proceedings of 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, October 1998.