Prioritizing Network Event Handling in Clusters of Workstations

Jørgen S. Hansen[1,⋆] and Eric Jul[2]

[1] SIRAC Project
INRIA Rhône-Alpes, France
jorgen.hansen@inrialpes.fr
[2] Department of Computer Science, University of Copenhagen, Denmark
eric@diku.dk

1 Introduction

The use of modern system area networking technologies [9,3] to construct tightly integrated clusters of workstations exposes two weaknesses of current operating systems. First, the low latency of current networks is often hidden from the application due to the high cost of interrupt handling. Second, network event handling during high load may result in serious performance degradation because all processor time is used for network event handling resulting in application starvation. This paper concerns the problems related to providing efficient and stable network event handling for clusters of workstations and network servers. By stable we mean that the throughput and response time of the system does not suffer when the workload offered to the system is increased beyond the maximum capacity of the system.

Our approach is based on assigning each network device a priority in the regular process priority range and allow events from the device to enter the system based on this priority. The handling of events from the device does not preempt processes with equal or higher priority, but will preempt lower priority processes. This integrates the processing of the stream of events from the device with the scheduling of regular processes thereby allowing for a natural batch processing of the events from the network adapter, and eliminating the risk of livelock. Making the devices visible to the operating system scheduler also allows us to eliminate the overhead of interrupts when the system is idle. When the scheduler detects that no process is runnable, it continuously poll the devices.

The rest of this paper is organized as follows. Section 2 explains our event handling approach in detail. Section 3 describes a prototype implementation in the Linux operating system, it's measured overhead, and the effects on a stream protocol for a system area network based on Scalable Coherent Interface (SCI). Related work is covered in Section 4, and our conclusions are drawn in Section 5.

⋆ The main part of this work was carried out as a Ph.D. student at the Department of Computer Science, University of Copenhagen

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 704–712, 2001.

2 Integrating Device Priorities with Scheduling

In this section, we present our device event handling based on device priorities and describe how it can be used to increase the stability of systems processing large amounts of network related events and to decrease network communication latency. We base our discussion on a priority-based scheduler using preemptive round robin scheduling of processes with the same priority. We first describe how our approach can be applied to uniprocessor systems, and then we look at the additional complexities of supporting multiprocessors.

2.1 Integrating Event Handling with Process Scheduling

In present day operating systems, the handling of events from a device usually has higher priority than any other task in the operating system. This can lead to degraded system performance when events arrive at a high rate [13,4] as time is spent on performing processing of arriving events instead of allowing applications or operating system to react to earlier events.

Instead of having a static high priority for device event handling, we propose viewing device event handling as a task whose scheduling is controlled by the operating system scheduler. In a priority based scheduler, this is done by assigning the device (or actually the device event handling) a priority in the priority range available for the scheduler. Thus, when events with a given priority are ready, one of the following three cases will occur: 1) processes with higher priority than the device are runnable, and the processing of events should be postponed, 2) processes with a priority equal to the device are runnable, and the processing of events should share the processor resources with these processes, and 3) no process with priority equal to or higher than the device is runnable, and the processing of events should take place immediately. From this, it is clear that when events are available for processing, a thread with the device priority dedicated to handling events by polling the device will behave as described in the three cases, and we therefore base our event handling on such polling threads.

2.2 Event Notification Using a Mix of Polling and Interrupts

Allowing the device polling thread to remain runnable even when there is no event to be processed obviously wastes processor resources. This problem can be avoided by having the thread block when there is no event available, and then using interrupts to schedule the process [13] when the first event occurs after a period of inactivity. However, the interrupt overhead is still incurred and the total latency experienced by the event will typically be higher. Instead, we use a combination of polling integrated with the operating system scheduler and interrupts to wake up a blocked thread. When there are runnable threads and they all have a priority lower than the device, we use interrupts to activate the polling thread, and in all other cases we rely on the operating system scheduler to poll the device. When there exists runnable processes, the scheduler polls all devices with priority equal to or lower than the highest priority runnable process

at the time when it needs to make a scheduling decision. Lower priority devices are polled to represent their polling threads in ready queues, if there are events to be processed. Thus, techniques such as aging can be used to prevent device starvation. When there are no runnable threads, we let the scheduler (or the idle thread) poll all devices continuously to allow the system to react quickly to new events. If processor power consumption is a concern, the processor can be halted (and interrupts enabled) after a certain period of time, e.g., a couple of minutes.

Polling I/O adapters across an I/O bus can be time consuming (relative to CPU-speeds) and should be avoided, if possible. Instead, we base our polling on event flags placed in physical memory. A network adapter that needs attention must raise its associated event flag to request kernel processing. On each scheduling decision, the scheduler checks these flags to detect pending events. This limitation on the signalling of events might seem restrictive, but most current high-speed networking technologies [3,6] include the necessary support for such event signalling. Conventional network designs could be supported through alternative polling handlers at a cost but having interrupts schedule the polling thread is likely to be more efficient.

One possible problem with postponing network adapter servicing is that the postponement may cause overflow of on-board buffers. If the network adapter supports the generation of high water mark interrupts, buffer overflow can be avoided by having these interrupts schedule a high-priority event handling thread. If, on the other hand, polling is used to increase stability, buffer overflow is how the increased stability during high load situations is obtained, i.e., by shedding network load as early as possible.

2.3 Priority-Based Event Handling on Multiprocessor Systems

Multiprocessor systems provide further opportunities for decreasing network latency through polling on idle processors because the many processors increase the probability of an idle processor. Additionally, compared to the limited form of prioritized distribution of interrupts supported in modern interrupt controllers [10], the use of polling threads allows a strict enforcement of the scheduling policy of the operating system. However, such interrupt controllers can be used to reduce the number of interruptions of high priority processes.

To review the added cost (in terms of additional processor synchronization) of priority-based event handling on multiprocessors, we revisit our two additions to process scheduling: polling of event flags and disabling and enabling of interrupts on priority changes. The polling of event flags need not be implemented as a critical region, as the worst case is that several idle threads activate the same polling thread. Enabling and disabling interrupts requires more careful handling than for single processors. The interrupts for a device should be enabled only when at least one processor is executing a thread with priority lower than the device and neither the device polling thread nor an idle thread is executing. If the scheduler uses a single run queue, the necessary global state is already maintained by the scheduler, but if a local run queue on each processor is used priority-based event handling adds synchronization to the scheduling loop.

3 An Example of Priority-Based Event Handling

We have implemented a prototype of the priority-based event handling for Linux 2.0 and 2.2. As the Linux scheduler implements a variation of priority based FIFO scheduling, the approach described in the previous section can be directly applied. The prototype has been used for the communication in a cluster of workstations connected by Dolphin's SCI cluster adapters [6]. These adapters provide hardware support for low overhead remote memory access through regular processor load and store operations as well as DMA. In the cluster, communication across the SCI network was handled by SciStream [8]—a TCP compatible stream protocol for SCI, and all nodes used 450MHz Pentium II processors.

In this section, we present an evaluation of the overhead added to the Linux scheduler and of the effects of priority-based event handling on SciStream communication latency and SciStream stability during high network load.

3.1 Added Scheduler Overhead

The overhead added to the Linux scheduler by (1) polling event flags and (2) changing interrupt status for devices was measured by augmenting the scheduler with measurements using the Pentium processor cycle counters. We implemented a kernel module 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. Changing interrupt status increments a counter, and thus reflects only the cost of invoking such a function. We used three configurations, where each configuration used from 0 to 16 allocated event flags with a size of four bytes each. In the **No Activity** configuration event flags are allocated but never raised. In the **Heavy Activity** configuration event flags are allocated and continuously raised. Finally, the **Interrupt** configuration forces a change in interrupt status on each scheduling decision.

We measured the overhead of the different configurations on a single processor system using Linux version 2.0. For **No Activity** the overhead amounts to 67 ns, and each additional flag adds an overhead of 25 ns. This is a bit more expensive than the cost of single memory reference suggested in Section 2.2, but our current implementation includes additional functionality such as partial masking of event flags. In the **Heavy Activity** case, the overhead is 135 ns for a single flag, and 91 ns are added for each flag. Finally, for **Interrupt** the overhead is 56 ns in the case with no allocated flags, and for a single allocated flag the overhead is 180 ns. Here each additional flag adds 39 ns of overhead. On multiprocessors, the priority-based event handling mechanism increases the overhead of each scheduling decision with the cost of taking and releasing a spin-lock.

We measured the cost of a context switch in Linux to be 4.2 microseconds. Thus, priority-based event handling 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. In most cases, the number of network devices in a single node will be small, and the overhead of supporting priority-based event handling will hardly be noticeable.

3.2 Latency of Event Handling

The priority-based event handling in SciStream is implemented using the remote memory access and remote interrupt facilities of the SCI cluster adapters. In the current implementation, raising the event flag is done by performing a remote fetch and increment, and throwing an interrupt (when interrupts are enabled) adds to this the triggering of a remote interrupt.

To establish the benefits of avoiding interrupts and the cost of using kernel threads for processing events, we compare the performance the priority-based event handling (**PriThread**) with interrupts (**Interrupt**) and application polling (**AppPol**) where the SciStream receive operation continuously check a single connection for incoming data. The effects of low priority compute-intensive jobs is determined through a ***XX*Hog** version for **PriThread** and **Interrupt**, where an application, that only yields the processor when preempted, is present.

We evaluated the configurations by measuring the average one-way latency of 10,000 request-response exchanges of a one byte packet on a uniprocessor system. **AppPol** results in a latency of 10.1 microseconds. **PriThread** adds 20.0 microseconds to this overhead due to the remote event notification, context switch, and selecting the proper process for execution. The two stage process of generating the remote interrupt in **PriThreadHog** adds another 32.6 microseconds to the latency. The newer versions of the SCI driver software allow this to be performed in a single remote operation resulting in a latency equivalent to **Interrupt**. Both **Interrupt** and **InterruptHog** are 20.1 microseconds more expensive than **PriThread**. Thus, using priority-based event handling results in a latency reduction to $\frac{20.0+10.1}{40.1+10.1} = 60\%$ when compared to interrupts. We achieve similar performance gains on a dual processor system.

In summary, we see that allowing an idle workstation to continuously poll it's network adapters can significantly decrease network communication latency.

3.3 High Load Behavior of a Web Server

The high load behavior of SciStream was examined using the Apache web server and the tool httperf [14]. Using httperf, we determined the maximum sustainable load for each configuration through a series of experiments where we gradually increased the offered fixed request rate. For all test cases, we were able to exceed the maximum sustainable load. We let the web server supply two different documents: a small document with a size of 1,622 bytes (**SD**), and a large document with a size of 56,257 bytes (**BD**). For both **SD** and **BD** we used the **PriThread** configuration described in Section 3.2 and **PriImm** that performs event handling immediately when an event flag is raised and therefore resembles the static high priority of interrupts. In the experiments, we issued one request per connection and the device and web server threads had the same priority.

The resulting reply rates (replies/s) for the Apache web server are shown in Figure 1. The **PriImm** configuration is able to provide the highest reply rate (471 replies/s for **BD** and 1,150 replies/s for **SD** as opposed to 466 replies/s and 1,066 replies/s respectively for **PriThread**) for both document types. This

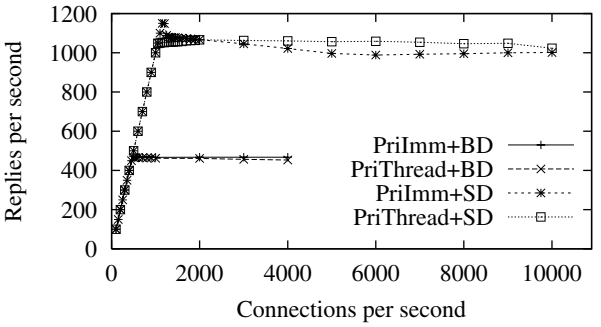

Fig. 1. Apache Reply Rates under High Load

is a result of the overhead of scheduling a polling thread in the **PriThread** configuration. Surprisingly, the **PriImm** configuration is able to sustain a rather high reply rate beyond the maximum sustainable load. This is the result of how a full connection request queue in the web server is handled in SciStream (see also Banga et al. [1]). While the queue is full, the client program will get a "connection refused" error for any connection attempts. However, as shown by the average connection times in Figure 2 the full queues delay the processing of all incoming requests considerably. Our Apache web server used 255 concurrent processes to service requests, and the scheduling of these concurrent threads also increase the average connection time of the **PriThread** configurations as the offered load is increased.

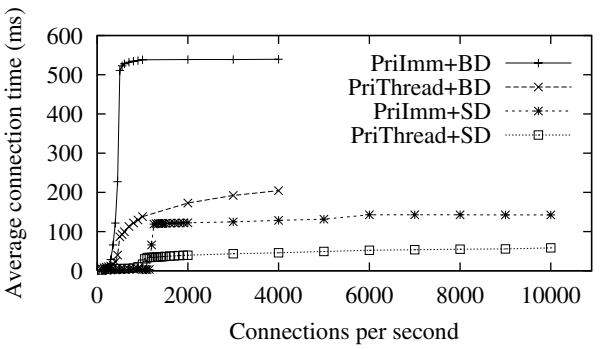

Fig. 2. Average Connection Time for Apache under High Load

Overall, we find that using priority-based event handling increases the stability (in the sense that the average response times are much lower during high network load) of the web server in the experimental setup.

4 Related Work

The work on eliminating livelock [13] explores the problem of livelock in great detail. A solution is suggested, where the interrupt is only used to schedule a polling kernel thread. The approach uses a kernel thread with higher priority than any user thread and can therefore result in thread starvation problems.

Polling and interrupts have been integrated with the thread management in a user-level thread library by Langendoen et.al. [11], but in contrast to our approach, they statically prioritize interrupts higher than thread processing.

Polling Watchdogs [12] consider polling network adapters when scheduling decisions are made, but they do not consider reducing the polling penalty by using flags in physical memory. Special purpose hardware is needed to keep the system responsive and to avoid network hardware buffer overflow.

Ensuring responsiveness when a network API does not support interrupts has been investigated by Perkovic et al. [15]. They perform low overhead network polling using event flags similar to ours but they mainly consider inserting this polling code into applications through source-code manipulation.

In scientific parallel programs, it might be beneficial to poll the network for a limited time before blocking, e.g., when a computation is followed by a data exchange. Damianakis et al. [5] successfully use fixed busy-wait thresholds before blocking on receive operations to reducing communication overhead.

Lazy receiver processing [7] makes each process perform most of the processing of its own network communication. This makes it easier to account for the time spent by each process on network communication. Preferably, the network hardware performs the packet demultiplexing onto the communication endpoints but if this is not possible a kernel thread handles the demultiplexing.

In signaled receiver processing [4], the protocol processing is also performed by the application, but the applications themselves can control whether the processing shall be performed synchronously, asynchronously or be suspended. Packet processing still relies on interrupts, and thus receive livelock may occur.

Resource containers [2] provide more accurate accounting of resource consumption for process-based operating systems through decoupling accounting information from processes. In our approach, resource containers can be used to improve the resource accounting for the device polling threads.

5 Conclusions

We have described a new approach to prioritizing event handling of network devices in a general operating system. A network device is assigned priorities in the scheduler priority range and by confining network event handling to a polling thread using this device priority, the processing of network events is integrated with the operating system process scheduling. The scheme relies on modifying the scheduler to perform part of the device polling. Our results show that this priority-based event handling increases the stability of network servers under high load, and, in addition, the scheme may reduce latency for lightly loaded systems.

Acknowledgements

Kåre Løchsen and Hugo Kohmann from Dolphin Interconnect (Norway) granted us access to the source codes for the PCI-SCI adapter and answered all of our questions. Povl Koch, Nokia and Emmanuel Cecchet, Simon Nieuviarts and Xavier Rousset de Pina, SIRAC project, provided us with valuable support for the SciOS prototype.

References

1. G. Banga and P. Druschel. Measuring the capacity of a Web server. In *USENIX Symposium on Internet Technologies and Systems Proceedings*, pages 61–71, 1997. 709
2. G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 45–58, February 1999. 710
3. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995. 704, 706
4. J. C. Brustoloni, E. Gabber, A. Silberschatz, and A. Singh. Signaled receiver processing. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000. 705, 710
5. 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. IEEE Computer Society Press, April 1–5 1997. 710
6. Dolphin Interconnect Solutions. PCI-SCI cluster adapter specification, May 1996. Version 1.2. See also `http://www.dolphinics.no`. 706, 707
7. P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *The Second Symposium on Operating Systems Design and Implementation Proceedings*, pages 261–276, October 1996. 710
8. J. S. Hansen, P. T. Koch, and E. 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, Rhodes, Greece, June 1999. ACM. 707
9. IEEE. *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE, 1992. Standard 1596-1992. 704
10. Intel Corporation. *Pentium Pro Family Developer's Manual. Volume 3: Operating Systems Writer's Guide*. Order Number 242691. 706
11. K. G. Langendoen, J. Romein, R. A. F. Bhoedjang, and H. E. Bal. Integrating polling, interrupts, and thread management. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 13–22. IEEE, 1996. 710
12. O. Maquelin, G. R. Gao, H. H. J. Hum, K. Theobald, and X. 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, 1996. 710
13. J. 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. 705, 710

14. D. M. Mosberger and T. Jin. httperf—a tool for measuring web server performance. In *Proceedings of the 1998 Workshop on Internet Server Performance*. ACM, 1998. 708
15. D. Perkovic and P. J. Keleher. Responsiveness without interrupts. In *Proceedings of the 1999 International Conference on Supercomputing*, June 1999. 710