A Scheduling Scheme for Network Saturated NT Multiprocessors

Jørgen Sværke Hansen and Eric Jul
Department of Computer Science, University of Copenhagen (DIKU)
Copenhagen, Denmark

A Scheduling Scheme for Network Saturated NT Multiprocessors

Jørgen Sværke Hansen Eric Jul

Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, 2100 Copenhagen, Denmark
E-mail: {cyller,eric}@diku.dk

Abstract

The use of high performance networking technologies, e.g., ATM networks, demands much from both operating systems and processors. During high network loads, user threads may be starved because the processor spends all its time handling interrupts.

To alleviate this problem, we propose using a two level network interface servicing scheme that uses interrupts during low network loads to provide low latency, and polling threads during high network loads to avoid user thread starvation.

In this paper, we examine the use of such a scheme on dual-processor workstations running Windows NT connected by an ATM network. Performance evaluation of our multiprocessor prototype implementation shows that using our two level scheme can improve performance when used carefully.

1 Introduction

High performance networks based on, e.g., ATM often demand substantial processor time; so much that processors can become saturated with network traffic leaving little or no time for actually processing data.

As part of a project concerning ATM network striping[1], we have considered how to efficiently handle multiple network interfaces. Processing the data that arrives on just a single high-speed network interface is a problem even for fairly high performance workstations; using several high-speed network interfaces (as we will be doing when performing network striping) will only aggravate the situation. Such processor overload can be handled by using extremely powerful processors. However, there are both economical and physical limits on how fast a processor that it is possible to use. As an alternative we propose using multiprocessors to provide sufficient processing power.

Previous work in the area of multiprocessor network performance has concentrated mainly on improving the performance of higher level protocols ([1], [4] and [5]), and furthermore, these approaches use a single network interface. We consider the scheduling issues related to handling one or multiple network interfaces on multiprocessors.

In the following, we first describe the problems of thread starvation, then we present a two level network interface servicing scheme that uses interrupt-driven servicing at low network loads, and polling threads at high network loads. Lastly we evaluate the performance of the two level scheme.

We have implemented this scheme on dual-processor workstations running Windows NT connected by an ATM network. As network interfaces, we use Olicom ATM network interfaces, and Olicom A/S has provided us with access to the source code for the ATM network interface drivers.

2 User Thread Starvation

In interrupt-driven kernels the total processor usage of a network application can be split into two parts, a part used by the user threads, and a part used by the interrupt-routine. Ideally, the partitioning should be such that the interrupt-routine delivers packets at the rate in which the user thread consumes them. To avoid packet loss in case of timing mismatch between user thread and interrupt-routine, a limited number of packets may be buffered in the I/O subsystem untill the user thread collects them. As long as the total demand for processing power does

[1]For more information visit the project homepage at http://www.diku.dk/distlab/Research/CIT/SPAN/span.html.

not exceed what is available, this should cause no problem.

When processing power is a problem during heavy network loads, the threads on the system are starved due to the fact that interrupt-routines have absolute priority over any other thread in the system, and thus the bulk of processing time is used processing interrupts from the network interfaces receiving data. The actual consumers of the received data are not allowed to process the data, and this may cause upper layer buffers in the network subsystem to overflow. The result can be that a large amount of processing power is used on receiving data that is subsequently discarded. Mogul and Ramakrishnan [3] identified this problem and propose to avoid this situation (which they call *receive livelock*) by using interrupt-initiated polling. When a network interface issues an interrupt, its handler merely starts a polling thread. This thread is scheduled together with any other threads in the system, thus reducing the livelock problem. Another benefit from using a polling thread is that the number of interrupts and context switches per received network data unit is lowered in stress situations.

The thread starvation problem is not necessarily removed by adding more processors to an interrupt-driven system. Because an interrupt steals processing time from the thread that it is interrupting, there is the danger of starving a thread during heavy network load. On a multiprocessor, a situation (which we call *thread pinning*) may arise where some processors are almost idle, while another processor is heavily loaded servicing the network interfaces and starving the thread that was to process the incoming data. Furthermore, in the case where a multithreaded user application is used, the user thread worklo709 V 1438 708 V 1441 707 V 1444 706 V 1447 705 V 1450 704 V 1453 703 V 1456 702 V 1459 701 V 1462 700 of the interrupt-routine. This may produce suboptimal performance as illustrated in figure 1. The figure shows how the processor usage is divided between user threads, interrupt-routine and idle time on a two processor machine. As full load is reached, the interrupt-routine starts stealing processor time from the user threads. This continues until the processor usage of the interrupt-routine reaches the capacity of a single processor. If the multiple threads are sharing data, the interrupt-routine might further degrade performance, if it interrupts a thread holding a lock to shared data. As illustrated in figure 2, this will occur when the total load generated by the user threads exceeds the capacity of one processor. From that point the user threads, that are sharing processor with the interrupt-routine, may be interrupted, and thus result

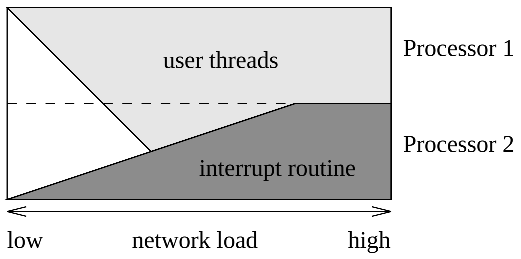

Figure 1: Processor usage as a function of network load on a two processor machine in the case with suboptimal performance during high load due to interrupt-handling.

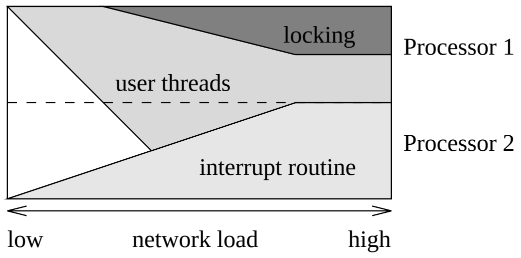

Figure 2: Processor usage as a function of network load on a two processor machine illustrating the performance degradation caused by user threads waiting on locks held by an interrupted thread.

in that threads running on the other processor must wait for a lock held by the interrupted thread. In the figure the idle time resulting from this is marked with *locking*.

To alleviate these problems, the scheduling of network handling on multiprocessors needs to be considered. One possible solution to the thread pinning problem is that the interrupt-handling routine at regular intervals yields the processor, thus allowing the user thread to be rescheduled—possibly on a less loaded processor. Another solution is to utilize that some multiprocessor architectures (e.g., the Pentium Pro) have support for controlling which processor is to receive a given interrupt on the basis of a priority assigned to each processor. As long as there is only one active thread handling network data, and as long as there are fewer network interfaces than processors, this would alleviate the thread pinning problem. Alternatively, one might consider simply disabling interrupts from the network interface(s) causing the overload, but this might hinder the progress of the user threads in the case where they depend on sending data as part of their processing. Lastly, a scheme with a polling thread as described in [3] can be used. By letting one or more threads handle the network interfaces, the usual scheduling mechanism of the operating system may be used to distribute the load on the available processors. This should be

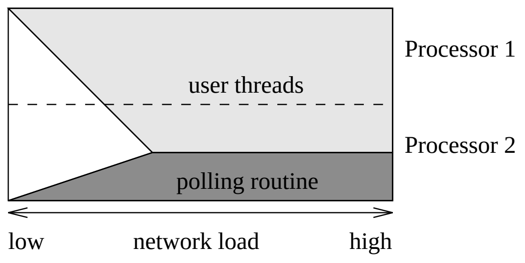

Figure 3: Processor usage as a function of network load on a two processor machine in the case where the use of a polling thread prevents performance degradation.

able to solve the problems regarding thread pinning, interrupt-routines stealing processor time from user threads and locking. This is illustrated in figure 3. Here the polling routine does not steal cycles from the user thread, and thus the processor time is used in a way that maximizes throughput. Furthermore, as there are no interrupts, threads holding a lock cannot be interrupted.

3 Our Two Level Scheme

The problems described in the previous sections lead us to abandon pure interrupt-driven network interface handling. The network handling based on interrupt-initiated threads seems attractive when the network load is high, but it would be nice to avoid the delay caused by both issuing an interrupt and making a context switch in order to process a packet when the network load is low. We therefore use a two level scheme where interrupt-driven servicing of the network interfaces is used until a certain level of network traffic, and above that level, a polling thread scheme is used.

4 Windows NT Implementation

This section provides a closer look at how we have implemented this scheme in Windows NT. First we give a brief description of the relevant parts of Windows NT I/O management as this provides the basis for the further discussion. Then we look at how to detect livelock in Windows NT, and finally we discuss how support for this scheme could be integrated with the current Windows NT I/O subsystem.

4.1 Windows NT I/O Management

In our description, we use a simplified model of a network protocol stack, where we have a transport driver placed on top of a device driver. In Windows NT, the layers interact by passing I/O Request Packets (IRPs) from one layer to the other. This is done via an I/O manager. In the following we describe how data transmission and reception is handled by this model.

On data transmission, the transport driver passes an IRP to the device driver, where the IRP is either processed by a device driver dispatch routine, or—in the case where the device is busy—queued for later processing. When the transmit operation is completed, the IRP is returned to the transport driver. This causes an I/O completion routine to be called in the transport driver. This I/O completion routine is often just a queuing of the IRP for further processing. In the case, where the completion of the transmit operation relies on a hardware interrupt from the device, the dispatch routine would return, and rely on an interrupt handler to complete the transmit operation. The NT interrupt handling consists of two steps - first the hardware interrupt causes the execution of an Interrupt Service Routine (ISR) running at device Interrupt ReQuest Level (IRQL), which does minimal work (e.g., disabling interrupts). This causes a Deferred Procedure Call (DPC) to be queued. This DPC is executed by a software interrupt when the IRQL drops below Dispatch/DPC Level (this is below device IRQL, but above normal thread execution level). This DPC handles the bulk of the processing.

When data is received on a device, the data is passed on to the transport driver in an IRP as described above.

The IRP queues can either be managed by the device driver or the I/O manager. Transmit operations are handled by a set of device driver dispatch routines. These may rely on interrupts to signal the completion of a transmit operation, and thus a part of the transmit handling is placed in the DPC.

4.2 Detecting User Thread Starvation

The main problem is to detect user thread starvation, i.e., when to make the transition between interrupt-driven and polled I/O. We consider the following possibilities:

Length of network data queues By monitoring the length of the network data queues (possibly IRP queues) that are emptied by the user thread, it should be possible to detect when user thread starvation occurs. The problem is that these queues are often internal to the transport driver requiring that the device driver has access to information about the size of the queues in the transport driver. As a transport driver may be bound to several devices, it should only be the IRPs belonging to the device that are reported back.

Interrupt rate By monitoring the interrupt rate of a device, an interrupt rate threshold value could be used to decide when the network load is high. The problem is that many device drivers use interrupt batching, i.e., processes multiple packets per interrupt.

Amount of time spent processing interrupts By measuring the percentage of processor time used by the DPC of the device driver, it should also be possible to detect user thread starvation. The measurement of the processing time is complicated by the fact that a DPC may be interrupted, but as the interrupts primarily are hardware related interrupts the impact should be negligible.

We have based our implementation on measuring the interrupt processing time as this is simple to implement. The transition from polled to interrupt-driven I/O is made when the polling thread lacks work to do.

4.3 Operating System Support

As the work done by the polling thread and the interrupt handlers is almost the same, it would be beneficial to integrate support for both interrupt handling and polling in the operating system. By letting a device driver register routines explicitly for polling threads and interrupt handlers, the I/O Manager can take active part in the decisions on what type of I/O handling to use , e.g., by monitor the execution time of interrupt handlers and initiate polling.

5 Results

In the following, we compare our two level scheme with a standard purely interrupt-driven device driver. In order to evaluate the viability of the two level scheme, we look at network latency, thread pinning, and finally we examine the effects of user thread starvation on multithreaded applications.

5.1 Methodology

To produce an overload situation on the receiving machine, a Dual Pentium Pro 200 MHz host was used as the transmitting side, and a Dual Pentium 133 MHz machine as the receiver. Both machines were running Windows NT 4.0 with service pack 3. The two machines were each equipped with two Olicom RapidFire 155 Mbps ATM adapters. All performance tests have been made using the TCP/IP protocol stack shipped with Windows NT on top of the Olicom driver configured to use Classical IP with a PVC between each pair of network adapters. As network load generator we use the network performance measurement tool `netperf`[2]. Again, to overload the receiver, we use the UDP protocol. In our two level scheme we used a threshold of 50%, i.e., the transition to polling was made, if an interrupt-routine used more than 50% of the processing time on a single processor for a period of more than two seconds.

5.2 Latency

To compare the overhead introduced by using the polling routine, we compare the latency of a pure interrupt-driven system, our two level scheme, and an interrupt-initiated polling routine. The interrupt-initiated polling routine is obtained by modifying the two level scheme implementation, so that all the DPC does is to signal the polling thread. The measurements are illustrated in figure 4, and show that the two level and pure interrupt-driven schemes have about the same latency, which is between 25 and 50 μsec higher than the interrupt-initiated polling scheme. Thus, the overhead of monitoring the execution time of the interrupt-routine is negligible, and low latency is achieved at low network load.

5.3 Thread Pinning

When thread pinning occurs, we expect to see two different levels on the rate of received data, one high level corresponding to the case where the user thread and interrupt-routine execute on different processors, and a low level in the case where they are executing on the same processor. We look at user threads running both at normal and real-time priority. User threads at real-time priority should be more vulnerable to interrupt-routines stealing cycles, as they can only be preempted by threads with higher real-time priority.

In figure 5 we show how thread pinning occurs on the receiving machine during a throughput measurement using 1024 bytes UDP packets at various send rates. In order to show the instability and variation in throughput, we conducted 10 measurements for a series of send rates for each priority, and show each of these measurements as a single dot in the graph. As can be seen from the figure, the received rate of the normal priority threads only split into two levels during extremely heavy load. This is due to the fact that only during maximum network load does the interrupt-routine run continuously and thus pre-

[2]Netperf can be obtained from The Public Netperf Homepage at http://www.cup.hp.com/netperf/NetperfPage.html.

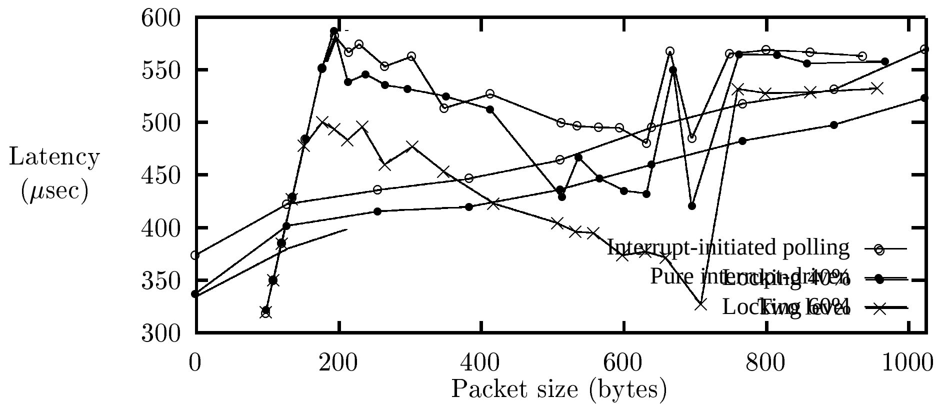

Figure 9: The effects of locking on throughput with constant workload. Each dot in the graph represents the average throughput of a series of measurements with 2048 bytes UDP packets.

In the Windows NT Kernel-Mode Driver Reference Guide [2] it is suggested, that periodic polling should be used to complete sends, when the total load on the processor, on which the interrupt-routine is executing, exceeds a certain level. This would also reduce user thread starvation, but does not address the problems of unevenly balanced load on multiprocessors.

7 Conclusion

We propose a two level device servicing system that uses interrupt handling during low network loads in order to provide low latency and polling during high network loads in order to prevent user thread starvation. This can be integrated in the I/O system of Windows NT.

Our current prototype on a couple of dual processor workstations running Windows NT 4.0 shows that the scheme is able to improve performance on network saturated multiprocessors.

Acknowledgments

We wish to thank Olicom A/S, in particular Tomasz Goldman and Kim R. Pedersen, for providing the necessary ATM hardware, for giving us access to their driver source codes, and for providing support in general. The project is funded in part by The Danish Natural Science Foundation and The Danish National Centre for IT Research.

References

[1] Mats Björkman and Per Gunningberg. Locking Effects in Multiprocessor Implementations of Protocols. In *Proceedings of SIGCOMM '93*, pages 74–83, September 1993.

[2] Kernel-Mode Drivers Reference Guide. Part of the Windows NT 4.0 Device Driver Kit, 1996.

[3] Jeffrey Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[4] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance Issues in Parallelized Network Protocols. In *Proceedings of OSDI '94*, November 1994.

[5] Gerald W. Neufeld, Mabo Robert Ito, Murray Goldberg, Mark J. McCutcheon, and Stuart Ritchie. Parallel Host Interface for an ATM Network. *IEEE Network*, pages 24–34, July 1993.

[6] Jonathan M. Smith and C. Brendan S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, pages 44–52, July 1993.