Embedded, Real-Time and Operating Systems Program

Maintaining End-system Performance under Network Overload

Luke Macpherson and Gernot Heiser

UNSW-CSE-TR-0412

March 2004

mailto:disy@cse.unsw.edu.au
http://www.cse.unsw.edu.au/~disy/
Operating Systems and Distributed Systems Group
School of Computer Science and Engineering
The University of New South Wales
UNSW Sydney 2052, Australia

Abstract

Network performance is currently outpacing the performance improvements seen by host systems, leading to a significant performance gap between the throughput which may be supported by a network interface, and the actual throughput which can be achieved by a typical end-system. Because this is the case, end-systems must be able to cope with applied loads which exceed their capacities. In particular, system performance in terms of latency, throughput, and jitter should not deteriorate under overload.

This paper evaluates the use of intelligent software-based control algorithms adjusting the interrupt-holdoff time and the available DMA buffer space in order to prevent receive livelock on commodity hosts and network adaptors. We present a simple analytical model of packet latency, which allows us to analyse system performance under overload.

The control algorithm has been implemented in the FreeBSD operating system. Experiments show excellent scalability under overload, comparing favourably with previous approaches. Furthermore, the implementation is less intrusive on operating system design than prior approaches with similar goals.

1 Introduction

Modern Gigabit Ethernet interfaces are easily able to cause receive livelock in current systems if those systems are not designed carefully. Despite the large amount of research into avoiding receive livelock and improving performance during overload, many systems continue to be susceptible to such performance problems. Given the recent release of 10Gb/s PCI Ethernet cards, this situation is likely to become even worse in the near future.

One reason why receive livelock is still seen in modern systems is that the existing methods for avoiding livelock require considerable programming effort to retrofit to an existing system, or require complicated network interface hardware. What is needed is a method of controlling livelock which is effective, but does not require significant modification to the operating system or additional hardware features.

In this paper we will propose a system for eliminating receive livelock and improving performance under overload conditions, which does not require custom hardware and can be implemented with only slight modifications to the device driver code, and TCP/IP stack.

2 Interrupt overheads and receive livelock

Gigabit Ethernet interfaces provide a significant challenge on current hardware because the packet rate, and hence interrupt rate, is potentially extremely high—on the order of one million packets per second. Because the overhead of servicing an individual interrupt is non-zero, there exists a maximum rate of interrupts at which all processor time is expended servicing interrupts, leaving no time remaining to process received packets. This situation is known as receive livelock, and is characterised by a degradation in achieved throughput as the received packet rate increases [8]. Therefore, it is necessary to reduce the overall impact of interrupt overheads in order to provide good performance with modern high-speed networks.

3 Past approaches

3.1 Interrupt-per-packet

Although this design is rarely used by modern network interface drivers, early network drivers existed in an environment where interrupt overhead was much lower than the inter-packet arrival time. This gave rise to a driver design where, upon receipt of an interrupt, the driver would dequeue a single packet from the network card, before returning from that interrupt.

For later systems, where inter-packet arrival times are close to, or even less than the overhead of an interrupt, an interrupt-per-packet design leads to excessive interrupt overheads, and increased packet latency.

3.2 Interrupt batching

Interrupt batching is a widely implemented attempt at reducing the excessive interrupt overheads associated with interrupt-per-packet designs [3, 8].

This technique is based on the observation that when inter-packet arrival time is close to, or less than, the overhead of an interrupt, multiple packets will arrive during the interrupt delivery period and be enqueued by the network card. Moreover, additional packets may arrive while earlier packets are being dequeued. Upon receipt of a single interrupt, a system which implements interrupt batching dequeues packets from the network interface until no packets remain to be processed.

Unfortunately, when the packet arrival rate is high, as soon as the driver clears the network interface's interrupt mask, a new interrupt is generated, resulting in the driver's interrupt handler being invoked again almost immediately. Because interrupts execute at the highest priority level, this design causes receive livelock to occur, as there is no time between interrupts for higher level processing of incoming packets to occur.

3.3 Polling

The traditional solution to excessive interrupt overheads is to avoid the use of interrupts entirely, in preference of a polling approach where the network interface is serviced periodically and any waiting packets dequeued.

There are many approaches to polling, each with different advantages and disadvantages. One of the difficulties encountered in discussing polling-based systems is that the performance characteristics of polling implementations vary widely depending on the frequency and regularity of individual device polls.

3.3.1 Software polling

The simplest form of software polling is the busy-wait polling scheme, where a tight loop is used to poll for incoming packets continuously. The advantage of busy-wait polling is that latency is minimised because interrupt delivery time and context switch overheads are avoided.

The major drawback of busy-wait polling is that it requires that a large percentage of CPU time be dedicated to servicing the device. This is not practical in systems where CPU time must be available for other tasks. As such, busy-wait polling is typically used by dedicated routers and similar devices, rather than more general-purpose computer systems.

Rather than using busy-wait polling, another approach to software polling is to poll the network interface at specific points during code execution, using software timed-intervals.

The advantage of this approach over busy-wait polling, or even using one interrupt per packet, is that packets can be dropped by the network interface, when the network interface's supply of receive buffers has been exhausted. This means that if the system fails to keep up with the applied load, and consequently polls

the device too infrequently, excess load is shed without increasing the processing time required to handle incoming packets. This gives a well implemented polling scheme the ability to resist receive livelock.

3.4 Hybrid interrupt and polling approaches

3.4.1 Polling watchdog

One of the potential problems with software-based polling schemes is that, in order to provide consistent and low latencies, they require either a high rate of polling, or a strict polling interval. Both of these requirements can be difficult to meet in software without incurring significant overheads.

One proposal designed to combat this problem is the polling watchdog [3, 7]. This timer is a feature of the network interface card, and generates an interrupt if the device is not polled within a specified time interval after the reception of a packet.

This design allows software polling to be implemented with loose timing constraints, while still providing a guarantee that the interface will be serviced within the specific time bounds.

3.4.2 Clocked interrupts

Another approach which avoids the difficulties of software-based polling is the use of clocked interrupts [6, 9]. This approach relies on a hardware timer that generates interrupts at regular intervals. Upon receipt of such an interrupt the driver polls the network interface for received packets.

Clocked interrupts provide a simple way of limiting the rate of interrupts, and hence preventing receive livelock. Their implementation is similar to that of a conventional driver where the work of dequeueing packets is performed in the interrupt handler, however their behaviour in operation is similar to polling with a fixed regular interval.

3.4.3 Eliminating receive livelock in an interrupt driven kernel

In a study of receive livelock in the Digital UNIX operating system [8], Mogul and Ramakrishnan detail a scheme developed to eliminate receive livelock. They propose two alternative methods for controlling the rate of interrupts. In the first, all packet processing is done in the interrupt handler, interrupts are disabled until packet processing is complete. In the second method interrupts are only used to initiate polling — when an interrupt is received, a flag is set causing the polling thread to service that device on it's next iteration. Load shedding is provided by disabling input from the network interface temporarily.

The idea behind these methods is to maximise the maximum loss-free receive rate (MLFRR) by reducing the overhead of interrupts, and to avoid livelock by

either processing packets to completion, or having the hardware drop the packets before any work has been done to process them.

3.4.4 Hybrid interrupt-polling for the network interface

Hybrid interrupt-polling (HIP) [3] reduces overheads associated with a purely interrupt-driven design by switching to polling once a load threshold is exceeded. The strategy then goes on to optimise performance by dynamically adjusting the polling interval according to the inter-packet arrival time.

The primary advantage of this strategy is that it minimises the latency of traffic without incurring excessive polling overheads. The disadvantage of the HIP control algorithm is that it does not take into account the effect of overload. Under overload the polling rate continues to increase until the minimum polling interval is reached. This results in degraded performance and potential receive livelock.

This disadvantage could be overcome by using some of the approaches taken later in this paper, such as controlling the input packet rate by adjusting the number of available receive buffers.

3.4.5 FreeBSD polling

FreeBSD utilises a polling scheme which falls between (and combines some of the advantages of) the periodic polling and clocked interrupt approaches. FreeBSD's polling implementation aims to minimise the impact of context switches caused by interrupts, which can become a serious performance bottleneck under high network loads. It does this by disabling device interrupts and polling those devices whenever a timer tick occurs, a system trap is entered, or the idle loop is running. This means that the device is only polled once the cost of entering kernel mode has already been incurred by another event.

For network applications, with a high frequency of send and receive system calls, the high rate of system calls results in a very high rate of polling. This allows the FreeBSD polling system to avoid latency penalties which would normally be incurred by polling on timer interrupts alone. It also means that the timer interval has very little impact on performance when the system is under network load, as system call frequency tends to be much higher than the timer frequency.

3.5 Interrupt coalescing

Many modern network interfaces have the ability to defer interrupt generation, allowing multiple packets to arrive before an interrupt is generated [1]. This feature goes by several names, including interrupt coalescing, interrupt suppression, and interrupt holdoff, all of which aim to amortise the cost of the interrupt over multiple packets.

3.5.1 Fixed interrupt-holdoff time

One of the simplest forms of interrupt overhead amortisation is fixed interrupt holdoff, which involves the introduction of a fixed delay between the arrival of a packet and the subsequent generation of an interrupt. This delay allows multiple packets to arrive and be handled on a single interrupt.

A typical value for a fixed interrupt holdoff is 100μs, which is the default value used by FreeBSD's *nge* driver. In some environments, this holdoff time is a significant component of the overall delivery time of the packet. For comparison, the forwarding latency of our HP ProCurve 2708 Gigabit Ethernet Switch is less than 2.5μs for 64-byte-packets [2].

Another way to think about a fixed interrupt holdoff time is that it places an upper bound on the number of interrupts per second that the device can generate. In the case of a 100μs holdoff time, the upper bound on interrupt frequency is 10kHz, regardless of the behaviour of the operating system. In reality the maximum frequency will be lower than this, since time before the first packet is received, interrupt delivery time and packet processing time will further increase the time between interrupts.

The disadvantage of using a fixed interrupt holdoff time is that latency is unnecessarily penalised under low load conditions by having an interrupt holdoff time which is too long, while throughput is compromised under high loads by having an interrupt holdoff time which is too short.

4 Dynamic interrupt-holdoff time

There are a number of problems with past approaches which leave scope for significant improvement. Many of the approaches consider only the maximisation of peak performance, without considering overload and receive livelock. Lazy receiver processing (LRP) [4] does address both performance and livelock, however it requires specialised hardware to eliminate livelock. Eliminating receive livelock in an interrupt-driven kernel [8] also addresses performance and livelock, however this approach requires significant modification to the structure and operation of the kernel, which is a barrier to its widespread adoption.

Instead of using a fixed interrupt holdoff time, it is possible to vary the holdoff at run-time according to other system parameters. Dynamically controlling the interrupt holdoff time avoids the tradeoffs involved in selecting a fixed interrupt holdoff time, thereby improving performance over a wide range of throughputs and system loads.

In addition to coalescing interrupts, interrupt holdoff can be used to cause the network interface to drop incoming packets in a similar way to that provided by polling at fixed intervals. This means that if more packets are received during the interrupt holdoff period than can fit in the available DMA buffers, the excess packets will be dropped by the network interface without incurring any further overheads. This ability to shed load effectively should allow a dynamic interrupt

holdoff approach to scale well under overload, and successfully avoid receive livelock.

Although it is possible to carefully tune a fixed holdoff time and static number of DMA buffers, such an approach requires the system to be tuned specifically for each application and system on which it is used. By using an adaptive control algorithm to vary both the holdoff and number of DMA buffers at run time, it is possible to optimise performance for the present operating conditions.

5 Model of performance under overload

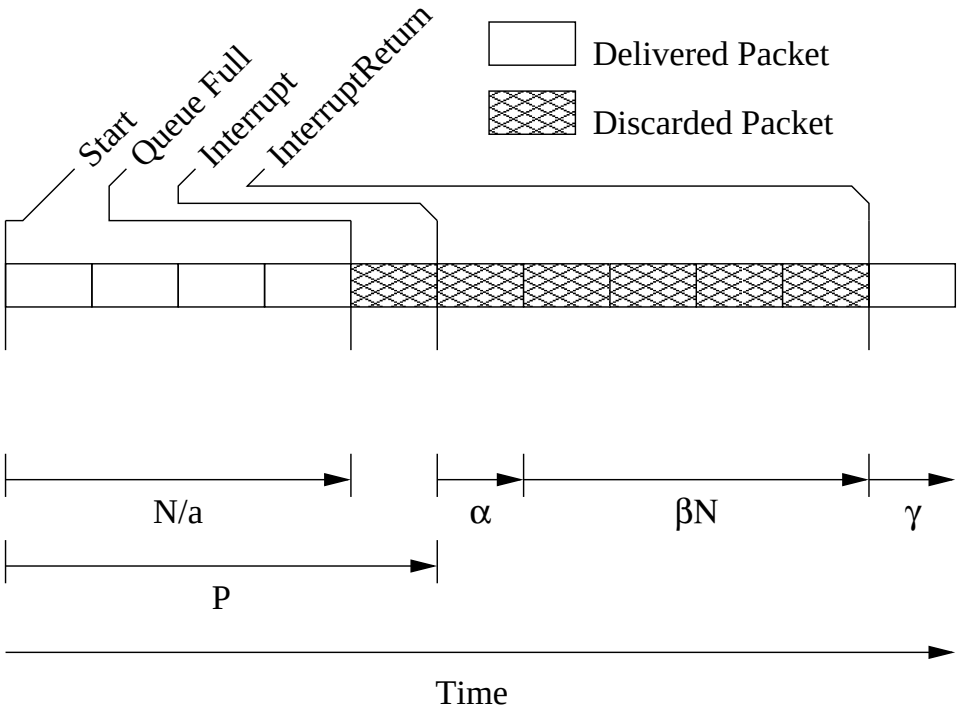

Figure 1: Packet handling event time line

In this section we will present a model for the performance of a system utilising interrupt holdoff under overload.

In the following equations,
α is the time to enter the interrupt handler
β is the time to dequeue one packet
γ is the time to return from the interrupt handler
N is the number of DMA buffers
P is the interrupt holdoff time, and
a is the applied packet rate.

The maximum achieved packet rate, r, is a function of P and N:

$$r = \frac{N}{P + \alpha + \beta N} \ . \tag{1}$$

The average packet latency, l, is a function of P, N, and a:

$$l = (P - \frac{N}{2a}) + \alpha + \frac{\beta}{2}N \ . \tag{2}$$

The overhead incurred by the interrupt handler, as a fraction of CPU time, c, is also a function of P and N:

$$c = \frac{\alpha + \beta N + \gamma}{P + \alpha + \beta N} \ . \tag{3}$$

Experiments show that $\alpha + \beta N$ is two orders of magnitude smaller than P, for useful values[1] of N and P. Determining exact values for α and β based on experimental results was not possible, as their effect is smaller than the measurement error. This is a fortunate result, as α, β are system dependent[2], while N, P and a do not vary between systems. Neglecting the insignificant contributions from these parameters gives us system-independent approximations of packet rate and latency:

$$r \approx \frac{N}{P} \ , \tag{4}$$

$$l \approx P - \frac{N}{2a} \ . \tag{5}$$

Figure 2 shows the approximate packet rate r, as a function of N and P for some reasonable values of N and P. It is important to remember that r is the maximum receive throughput of the network interface — if r exceeds the maximum throughput of the system, livelock may still occur. In effect, r controls the maximum applied load seen by the operating system.

6 Implementation

6.1 Basic approach

Our implementation of dynamic interrupt holdoff in the FreeBSD operating system consists primarily of modifications to the *nge* Gigabit Ethernet driver[3]. The changes are relatively minor, and add only 91 lines to the driver code.

[1] Useful vales of N and P are integer values which result in a packet rate r which is less than the maximum packet rate supported by Gigabit Ethernet, and which do not incur excessive packet latencies.

[2] α, β and γ are dependent on both the system hardware being used, and on the details of the operating system implementation, while N and P are parameters which can be freely chosen by the system, and can be externally controlled.

[3] For convenience, the socket layer was also modified to increment a counter whenever a packet is dropped —this is a one-line change, and is a simple alternative to having the kernel traverse the socket queues.

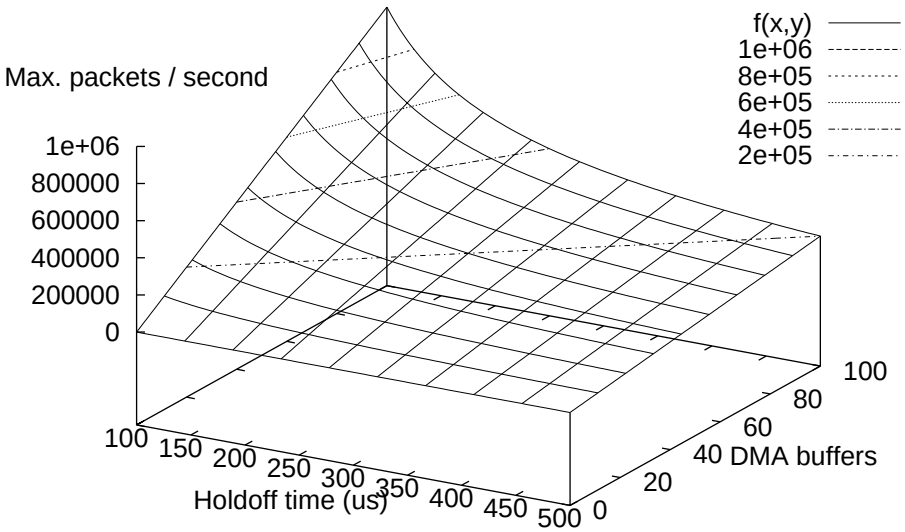

Figure 2: Maximum packet rate as a function of interrupt holdoff time and number of DMA buffers

The implementation has two important, but distinct goals; the first is to maximise the maximum loss-free receive rate of the system (MLFRR), and the second is to ensure that performance does not degrade once the applied load exceeds the MLFRR.

In order to maximise the MLFRR, we need to consider the performance of the system when it is not under overload. In this case, the system should not experience any packet loss, and the number of DMA buffers should be large enough that the throughput achievable by the network interface exceeds the throughput which can be achieved by the rest of the system. When the MLFRR has not yet been reached, average latency is proportional to interrupt holdoff time, while CPU utilisation is inversely proportional to the interrupt holdoff time.

Our strategy, therefore, is to minimise latency at low throughputs, while limiting CPU utilisation at high throughputs. We do this by using a low interrupt holdoff time at low throughputs, which increases with load until the MLFRR is reached.

Once the MLFRR has been reached, our goal shifts to minimising any degradation in performance. To this end, we control the number of DMA buffers, which according to Eq. 4 will limit the packet rate seen by the operating system.

The pseudo-code in Figure 3 provides a simplified version of the code utilised by the modified *nge* Gigabit Ethernet device driver to control the interrupt holdoff time and number of DMA buffers.

```
nge_adjust_holdoff(){
  static int backoff = 0;
  int        time_in, time_total;

  /* calc time in and out of kernel */
  t_in    = ...;
  t_total = ...;

  /* adjust interrupt holdoff time */
  offset = time_total - time_in - holdoff;
  holdoff = (((100-LOAD)*time_in)/LOAD)
             - offset;

  /* adjust number of DMA buffers */
  if(dropped_packets > 0){
      decrease_available_dma_bufs();
      dropped_packets = 0;
      backoff = BACKOFF;
  }else if(--backoff < 0){
      increase_available_dma_bufs();
      backoff = BACKOFF;
  }
}
```

Figure 3: *Pseudo-code describing the operation of the control algorithm*

6.2 Holdoff control

The time in microseconds is sampled when the driver's interrupt handler code is invoked, and again immediately prior to the interrupt handler returning. The difference between these two measurements is taken, giving the time spent inside the driver code servicing the network interface. This method cannot account for the additional overheads associated with taking the interrupt, such as the time to trap into the kernel and return from interrupt, as well as the cache miss costs resulting from cache pollution by the driver. Hence a fixed offset was added to the time spent in the driver code. This offset was experimentally determined to be 12μs on our development machine, but can be expected to be highly platform dependent. It would be possible for the system to determine the approximate size of that offset at startup time, but we did not implement this.

Once the time between interrupts and the time spent in the interrupt handler is known, it is possible to calculate the percentage of time spent in the interrupt handler, and to subsequently adjust the interrupt holdoff time to correct for any deviation from the desired value.

We do this by assuming each interrupt will take approximately the same time to process as the previous interrupt. The interrupt holdoff time is adjusted to give the correct ratio between time spent inside the kernel and time spent outside the kernel.

6.3 Rate limiting

In addition to dynamically controlling the interrupt holdoff time, the driver was modified to control the number of DMA buffers available for the receipt of incoming packets. By observing when packets are dropped within the kernel, both at the protocol stack's input queue and at the socket layer's input queue, it is possible to detect the onset of interrupt livelock.

The number of DMA buffers is controlled such that the rate of incoming packets does not exceed the rate at which packets are dropped by the operating system kernel. This approach ensures that system performance does not degrade once the maximum packet rate has been achieved.

The main caveat applying to this control scheme is that a misbehaving application could cause the network driver to drop packets even though the system was not experiencing overload. For example, an application could open a UDP socket and begin listening for packets, then cause a remote host to send packets to that socket, and cease issuing reads to that socket. The socket's receive queue would become full, resulting in packet-loss which is not associated with genuine overload. This problem could be solved in a production system by measuring the fraction of all packets which are dropped by the system (across all open sockets), and tolerating a certain amount of packet loss.

7 Evaluation

7.1 Experimental setup

Benchmarks were performed using *FreeBSD 5.1-release* on an Intel Xeon 2.66GHz processor, with 1GB of RAM. The processor's hyperthreading functionality was disabled for all benchmarks because it was unsupported by the FreeBSD polling implementation used. The Ethernet card used was a LinkSys EG1064, based on the National Semiconductor DP83820 chip, and was connected via a 64-bit, 33MHz PCI bus. This network card supports interrupt holdoff to be specified with a 100mus granularity (which is relatively course, see Section 3.5.1).

Load generation and performance measurement was achieved using seven 2GHz Celeron-based machines, connected via a HP ProCurve 2708 unmanaged switch. These machines were controlled using the *ipbench* distributed benchmarking utility [5], which is able to simultaneously measure latency and CPU utilisation at varying throughputs.

A UDP test was chosen for these benchmarks because it provides a non-responsive flow, and hence it is possible to see the behaviour of the machine under overload more clearly than if a responsive protocol such as TCP is used.

During this benchmark, a number of machines generate UDP packets which are directed at the machine under test. The test machine must deliver the packets to a user-level program which echos each packet back verbatim to the host from which the request originated. This task requires a minimum of two system calls per packet: a receive system call followed by a send system call. This represents a very high rate of system calls per second, and further increases the load on the machine being tested. It is also the scenario for which the FreeBSD polling scheme is designed to perform well.

7.2 Results for standard platform

Figure 4 shows the relationship between achieved throughput and applied load for a number of system configurations.

The effect of interrupt livelock when no interrupt mitigation is used is quite pronounced. 0μs interrupt holdoff reaches only a few hundred megabits per second before maximum performance is reached, after which performance degradation becomes severe.

The unmodified *nge* driver's use of 100μs holdoff achieves considerably better throughput before reaching livelock, and there is a gradual performance degradation as applied load is increased; however in this benchmarking scenario the network interface becomes saturated long before receive livelock becomes a problem.

FreeBSD's hybrid interrupt/polling implementation achieves a comparatively good maximum throughput. Under overload conditions the polling implementation's throughput shows gradual degradation as applied load is increased. This performance degradation can be attributed to a lack of effective load shedding by

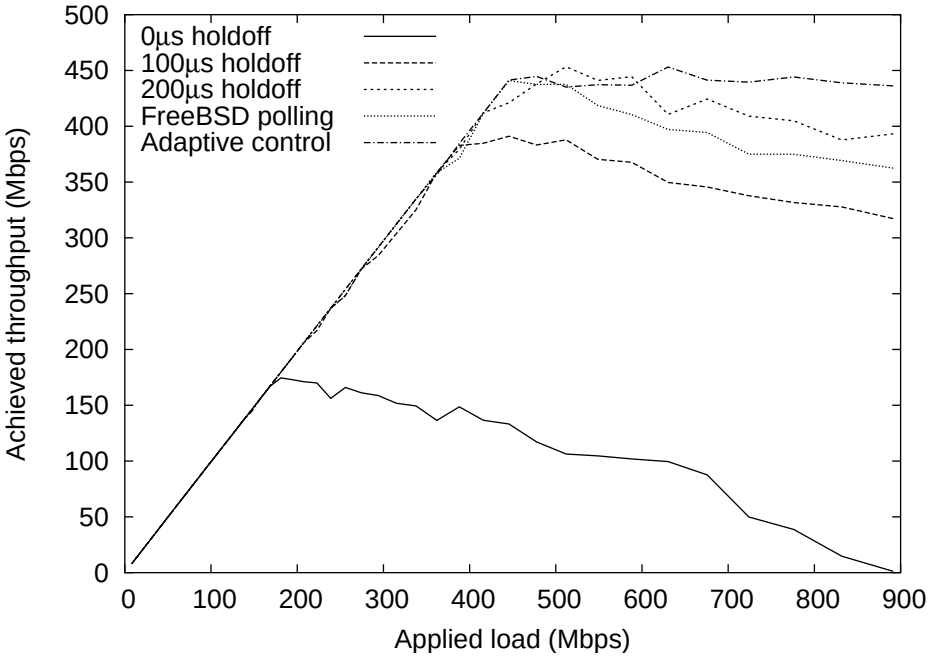

Figure 4: Achieved throughput vs applied load for 1024 byte UDP packets

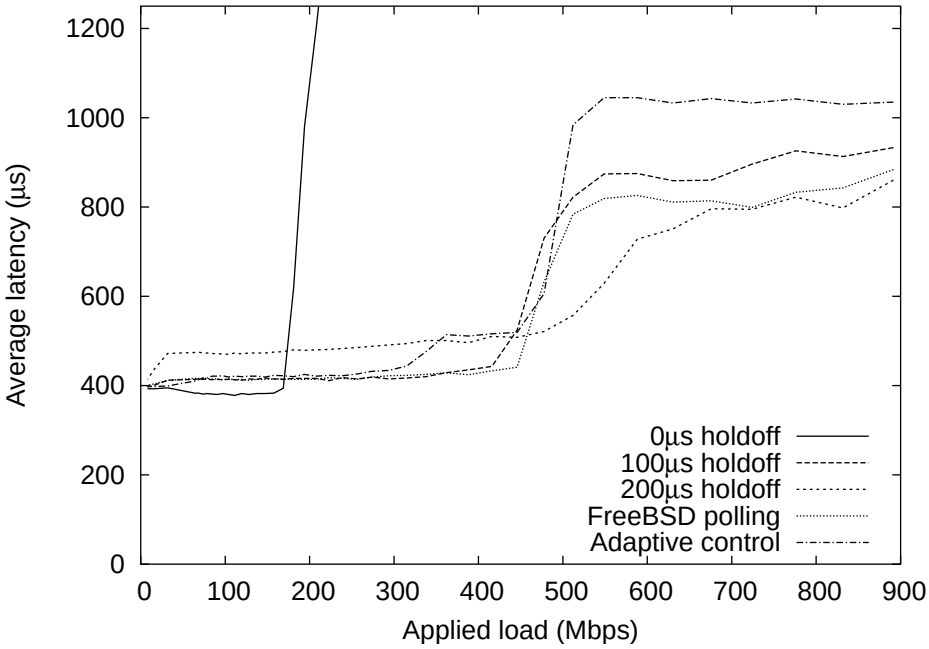

Figure 5: Average round-trip latency vs applied load for 1024 byte UDP packets

FreeBSD's polling implementation—because of the high polling frequency, packets are discarded by the device driver once the network protocol stack's input queue becomes full, rather than being discarded by the device itself. This means that as the applied load increases, the work which must be done by the driver in order to discard excess packets also increases.

The adaptive control implementation achieves a peak throughput almost identical to FreeBSD's polling implementation before overload is reached, however there is almost no degradation of throughput under overload. This improvement is attributed to the ability of the adaptive control algorithm to shed load by reducing the number of DMA buffers, and reduce interrupt-related overheads by increasing the interrupt holdoff time.

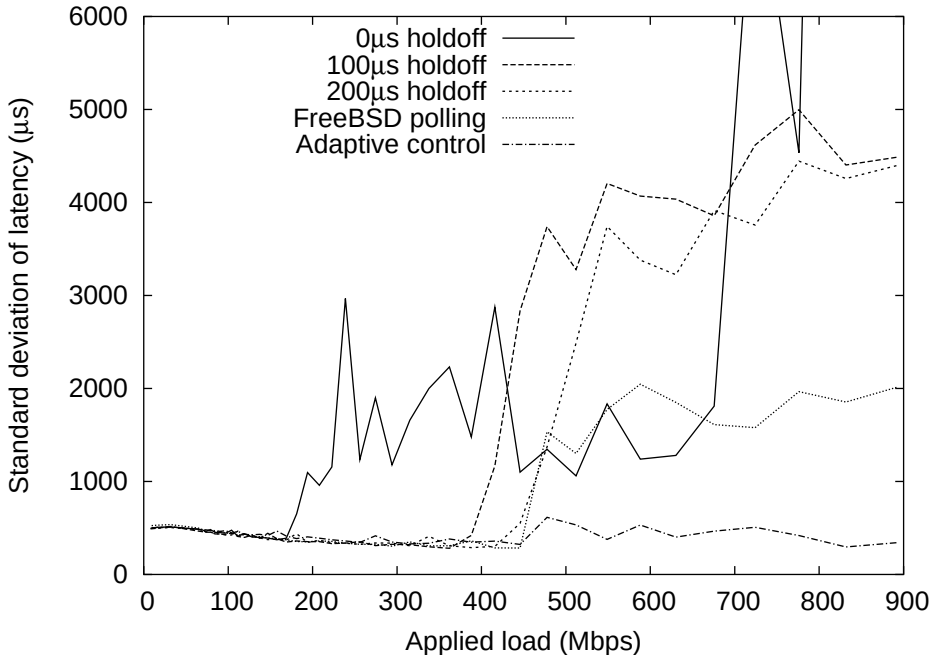

Figure 6: Standard deviation of round-trip latency vs applied load for 1024 byte UDP packets

Figure 5 shows the relationship between round trip latency and applied load for this same set of system configurations.

With low loads, 0μs holdoff results in the best case for average latency, however at the point where overload is reached, latency increases dramatically. 100μs holdoff gives a small increase in average round trip latency until overload is reached. Once overload is reached, latency is roughly doubled before continuing to scale well.

The FreeBSD polling implementation performs similarly to 100μs holdoff, but continues with low latency to slightly higher throughput, before exhibiting slightly lower average latency under overload.

Interesting to note in Figure 5 is that the average latency for the adaptive control implementation is somewhat higher (about 20%) than that of the default driver with 100μs holdoff and the FreeBSD polling implementation. Figure 6, however, shows that the standard-deviation of latency, which is a measure of jitter, under overload is much lower for the adaptive control implementation than that achieved by other implementations. Furthermore, the adaptive control could be tuned to trade latency against throughput, but we did not explore that possibility.

This effect is not only a result of changing the number DMA buffers or controlling the interrupt-holdoff time, but also due to a subtlety of the implementation of the adaptive control algorithm. The default FreeBSD driver, like most modern Ethernet drivers, uses interrupt batching, Furthermore, the FreeBSD polling implementation frees a DMA buffer as soon as the packet has been dequeued from that buffer. This means that new packets can continue to be received as soon as a packet is dequeued by the driver. While this approach results in lowered average latency, as some packets will be dequeued almost immediately after reception, it also has several negative effects.

Firstly, jitter is increased greatly under overload, because some packets wait for almost all of the interrupt holdoff time to be dequeued, while others are dequeued almost immediately. Secondly, because the number of packets received per interrupt is not restricted, degradation of performance under overload occurs more quickly. Given a high-enough applied load, the network driver may never return from the interrupt handler. Instead, the network driver could spend all its time dequeueing packets from the network card, and enqueueing them at the protocol stack.

The adaptive control algorithm was implemented in a way that minimises this effect. By only freeing DMA buffers immediately before returning from the interrupt handler, the algorithm prevents the network card from enqueueing further packets while servicing the interrupt.[4]

Figure 7 shows the relationship between CPU utilisation for the entire system, and applied load. Once again, 0μs holdoff performs significantly worse than the other approaches. This is due to the unregulated interrupt frequency.

Meanwhile, 100μs holdoff, polling and adaptive control all give remarkably similar results. Interestingly, polling starts with lower utilisation than both the fixed and variable holdoff schemes, but once overload is reached, tends to use more CPU time.

The adaptive control implementation gives the reverse effect, with slightly worse CPU utilisation at lower throughputs, while at higher throughputs it outperforms other approaches. The large dip in CPU utilisation around 350Mbps applied load is caused by switching from 100μs to 200μs holdoff. Lowered CPU utilisation at high throughputs is highly desirable, as it frees CPU time when it is most needed. This can be seen in Fig. 8, which shows the processing overhead per

[4]When the system is not experiencing overload, there are enough free DMA buffers to allow the system to continue to receive packets while the interrupt is being serviced.

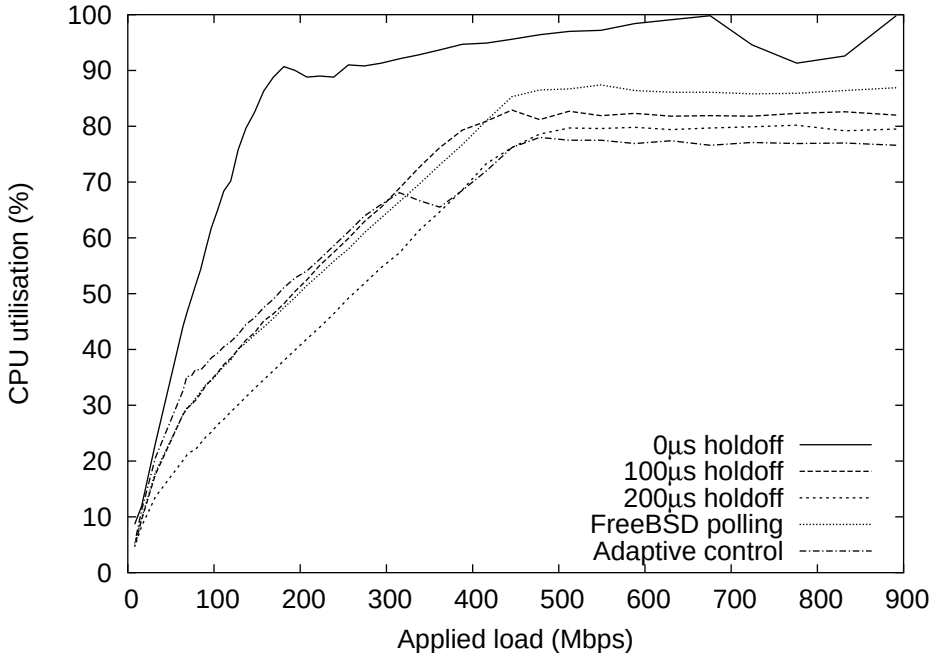

Figure 7: CPU utilisation vs applied load for 1024 byte UDP packets

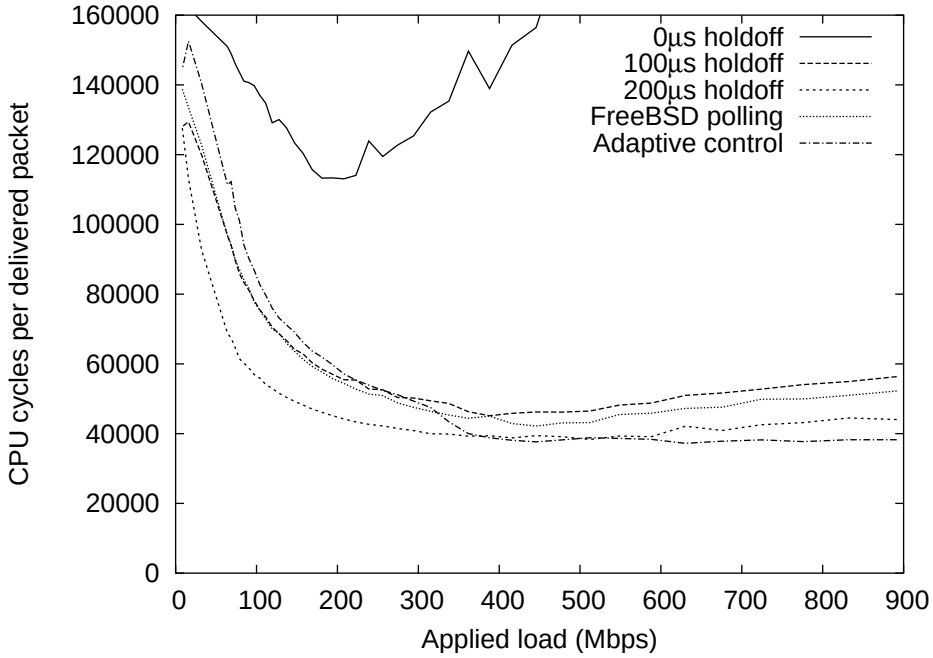

Figure 8: CPU cycles per delivered packet vs applied load for 1024 byte UDP packets

packet. The overhead of the adaptive scheme is at low loads very slightly higher than that of the 100μs holdoff or FreeBSD polling, but under high load is lowest and load independent, while for the other schemes the overhead increases with applied load.

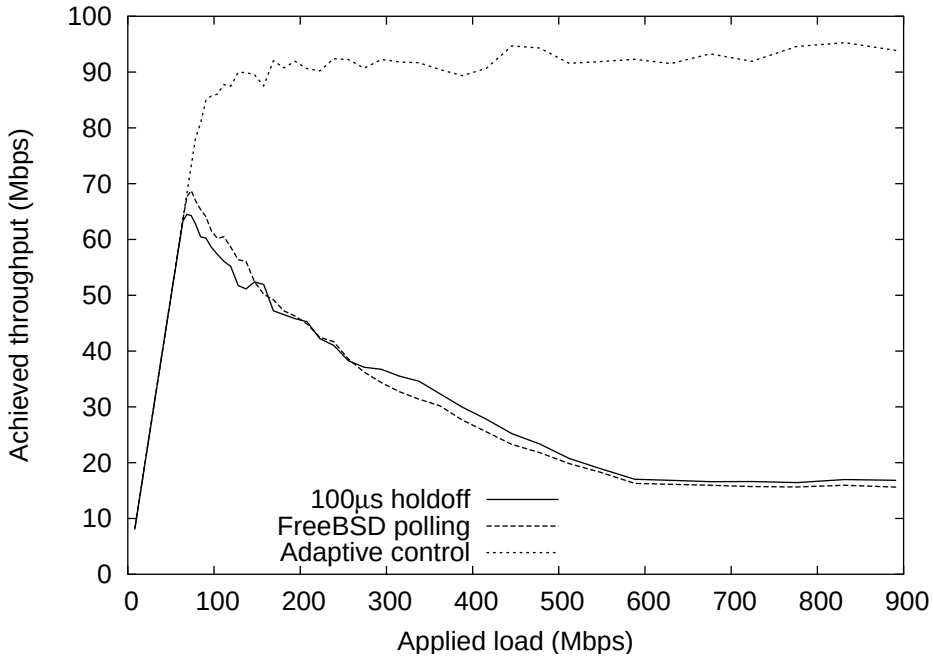

Figure 9: Achieved throughput vs applied load for 1024 byte UDP packets, slow machine

7.3 Simulation of faster networks

In the above experiments only the zero-holdoff case experienced receive livelock, all other schemes avoided this extreme degradation. This can be expected to change when faster network interfaces become available (or on embedded devices with lower processing capabilities). In order to simulate the effect of faster network adaptors we re-ran the benchmarks on a much slower machine, based on a 550MHz Pentium-III processor. This machine featured a 32-bit, 33MHz PCI bus and was equipped with a DLink DGE-500T Gigabit Ethernet card, also based on the DP83820 chipset.

Fig. 9 shows that the performance of the 100μs holdoff and polling schemes deteriorates dramatically as soon as the applied load exceeds 100Mbps, while the adaptive scheme improves until around 200Mbps, after which the maximum achieved throughput is maintained. A consistent picture emerges from the graphs of latency (Figs. 10 and 11) which both show clear degradation of the fixed holdoff and polling schemes, while the adaptive scheme is able to deal with the overload.

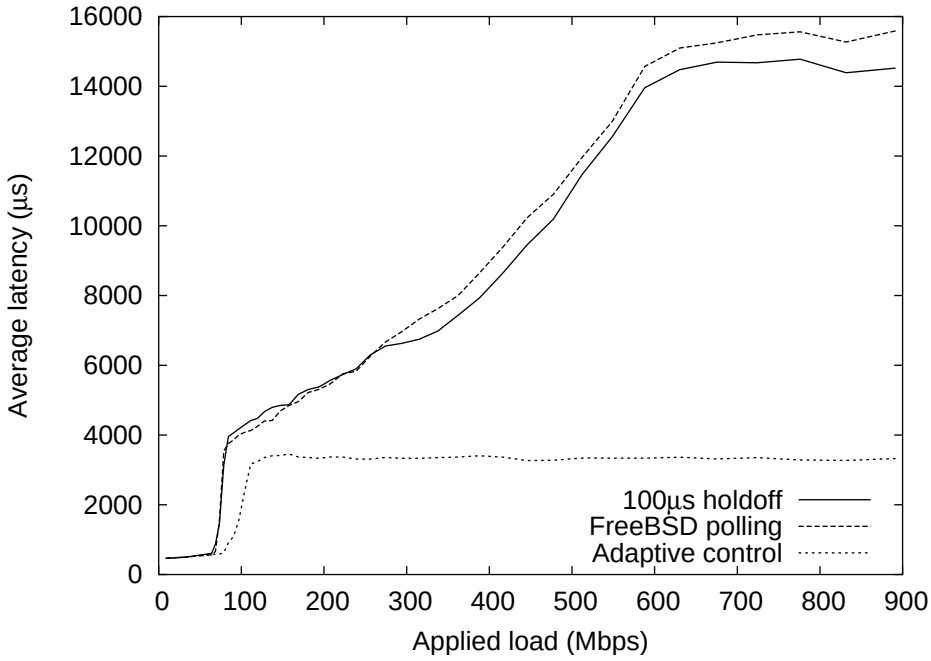

Figure 10: Average latency vs applied load for 1024 byte UDP packets, slow machine

8 Conclusions

Our study confirms that software-based schemes are able to defer or prevent receive livelock in a system overloaded by incoming network traffic. However, we also show that fixed schemes, such as fixed interrupt holdoff times or the hybrid polling approach implemented in FreeBSD, cannot satisfy the conflicting requirements of minimal latency under low load and maximisation of throughput under high load.

We have presented an adaptive scheme, which adjusts both the interrupt holdoff time and the available DMA buffer space to the network load. We have demonstrated that this scheme is able to maintain maximum performance under high load, without adding significant overhead under low load. Therefore, dynamic control of interrupt holdoff time is able to provide near-optimal performance over the full range of system load, rather than being optimised for a specific load point.

Importantly, the adaptive scheme is simpler to implement than regular polling, as it can be implemented almost entirely within the device driver — the kernel infrastructure required by a polling approach is not required.

While we can conclude that the control of receive livelock is possible with present network hardware, we can also observe that the 100μs interrupt holdoff granularity provided by the DP83820 chipset is too coarse. Useful values seem to range from 0μs to 1000μs, higher values would only be needed for systems with very high interrupt delivery costs. An interrupt holdoff granularity of 10–20μs would allow the control algorithm to adjust better particularly at low to intermedi-

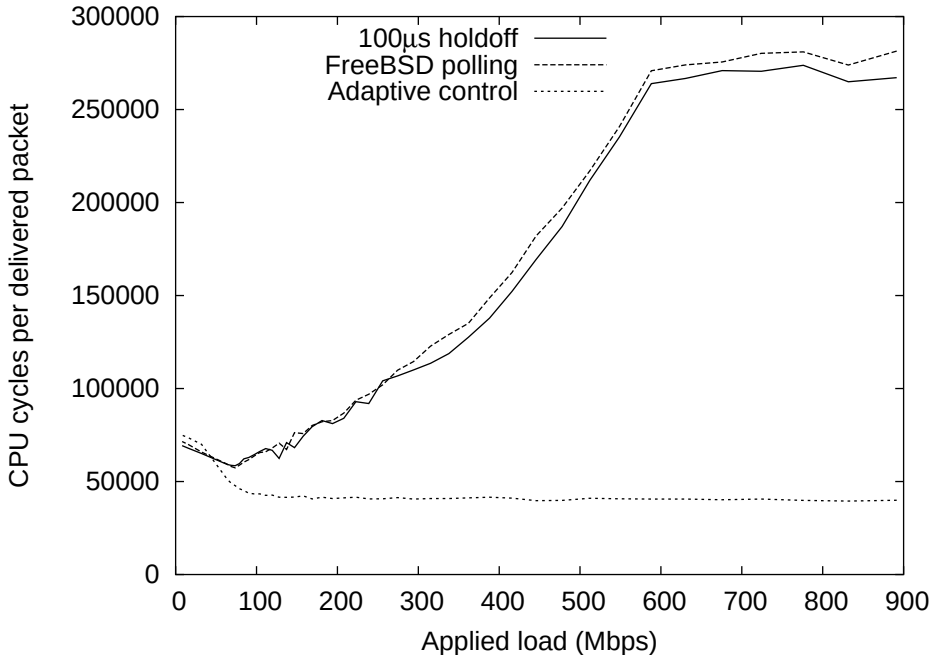

Figure 11: Cycles per packet vs applied load for 1024 byte UDP packets, slow machine

ate loads.

References

[1] Jeffrey S. Chase, Andrew J. Gallatin, and Kenneth G. Yocum. End system optimisations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[2] Hewlett-Packard Company. HP ProCurve 2700-series datasheet, October 2002.

[3] Constantinos Dovrolis, Brad Thayer, and Parmeswaran Ramanathan. HIP: Hybrid interrupt-polling for the network interface. *ACM SIGOPS Operating Systems Review*, 35(4):50–60, October 2001.

[4] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Operating Systems Design and Implementation*, pages 261–275, 1996.

[5] ipbench — a distributed framework for network benchmarking. `http://ipbench.sourceforge.net/`.

[6] Ilhwan Kim, Jungwhan Moon, and Heon Y. Yeom. Timer-based interrupt mitigation for high performance packet processing. In *5th International Conference on High-Performance Computing in the Asia-Pacific Region*, 2001.

[7] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling watchdog: Combining polling and interrupts for efficient message handling. In *The 23rd Annual International Symposium on Computer Architecture*, pages 179–188, 1996.

[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, 1997.

[9] Jonathan M. Smith and C. Brendan S. Traw. Operating systems support for end-to-end Gbps networking. *IEEE Network*, 7:44–52, February 1993.