Small Packet Traffic Performance Optimization for 8255x and 8254x Ethernet Controllers

Application Note (AP-453)

Revision 1.0
September 2003

Revision History

Revision	Date	Description
1.0	Sep 2003	Initial release.

Contents

Figures

Tables

Note: This page is intentionally left blank.

1.0 Introduction

1.1 Scope

This document discusses the "small packet" performance of Intel® 8255x and 8254x Ethernet controllers, and describes techniques for improving this performance through driver design optimization and configuration.

1.2 Reference Documents

The reader should be familiar with network device driver design and TCP/IP protocol. In addition, the following documents may be useful:

1. 82546EB Gigabit Ethernet Controller Networking Silicon Developer's Manual, Revision 0.75, Intel Corporation.
2. 10/100 Mb Ethernet Family Software Technical Reference Manual, Revision 2.1, Intel Corporation.
3. Interrupt Moderation using the 82540EM, 82545GM and 82546EB Gigabit Ethernet Network Controllers, Application Note 450, Revision 1.0, Intel Corporation.
4. Benchmarking Gigabit Network Adapters (Whitepaper), revision 1.0, Intel Corporation.
5. PCI: Efficient Use (Whitepaper), Intel Corporation (http://www.intel.com/support/chipsets/PC1001.HTM).
6. Beyond Softnet, J. Salim, R. Olsson, A. Kuznetsov (http://www.cyberus.ca/~hadi/usenix-paper.tgz).

Note: This page is intentionally left blank.

2.0 Performance Defined

Performance may have different meanings for different applications. Performance usually includes some or all of the following elements:

- **Throughput**. a measure of how much data an Ethernet device can transmit or receive per unit of time — typically measured in megabits per second (Mb/s). This is the most common metric. Larger throughput numbers are usually desirable.
- **Packet rate**. the number of packets that an Ethernet device may send or receive per unit of time — typically measured in packets per second. Larger packet rates are usually desirable.
- **Packet loss**. measures the number of packets lost en route, usually due to network congestion. Smaller packet losses are usually desirable.
- **Per-packet latency**. the delay associated with processing an incoming or outgoing packet. This is usually a function of processing power and efficiency of the local host, rather than any characteristic of the network itself. Smaller latencies are usually desirable.
- **Response time**. the delay between sending a packet on the network and receiving a response to or acknowledgement of that packet. Smaller response times are usually desirable.
- **Efficient resource utilization**. resources will include CPU utilization, memory utilization and bus utilization.

The above elements are related in many instances. For example, non-zero packet losses can result in two different packet rates for a single, unidirectional connection (i.e., transmit packet rate and receive packet rate). Ideally, these two packet rates will be identical — however, as packet losses increase, the receive packet rate may decrease while the transmit packet rate may remain nearly constant.

As different applications may require different types of performance, not all of the issues and suggestions that follow are appropriate for all applications.

Note: This page is intentionally left blank.

3.0 Small Packet Performance Issues

Limitations and tradeoffs are inherent in small packet traffic. These issues will affect any Ethernet implementation, regardless of link speed, host configuration or vendor.

3.1 Tradeoffs Inherent in Small Packet Performance

Optimizing for specific traffic patterns (in this case, small packet traffic) can often degrade other performance areas. Possible adverse effects are:

- **CPU utilization**. Software may need to acknowledge and process incoming packets more quickly, thereby requiring more CPU interrupts and causing an increase in overall system utilization. This is likely the most important tradeoff issue.
- **Resource requirements**. Software may need to allocate more resources in order to accommodate high rates of small packets.
- **PCI/PCI-X bus utilization**. An Ethernet controller may need to transfer packets across the local bus more quickly, increasing bus utilization and contention.
- **Network utilization and congestion**. The increased traffic rates that accompany small packet traffic will place greater demands on the network infrastructure.
- **Full-size packet performance**. Increases in small-packet performance may come at the cost of decreased full-size packet performance.

3.2 Network Protocol Overhead

The Ethernet protocol supports frame sizes from 64 to 1518 bytes in length (in the absence of packet tags and jumbo frames). In addition to payload data, each frame contains control information used to delimit, route and verify the frame (among other things). This extra

information consumes a portion of the available bandwidth, and reduces the potential throughput of the Ethernet. Likewise, protocols such as TCP/IP also introduce additional overhead (see Figure 1).

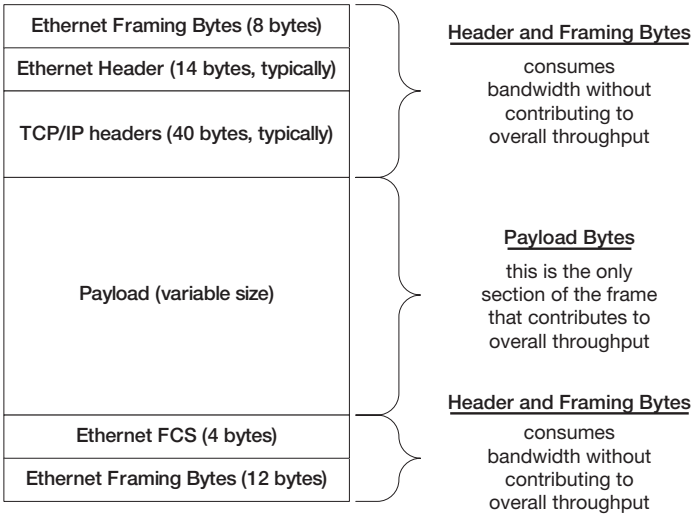

Figure 1. Typical TCP/IP Ethernet Frame

As frame size decreases, protocol overhead can occupy a significant portion of the network bandwidth consumed. Table 1 shows the effective unidirectional throughput of a Gigabit Ethernet device for minimum and maximum frame sizes.

Table 1. Throughput Calculations for Unidirectional Gigabit Traffic

	Minimum-sized "Raw" Ethernet frames	Minimum-sized Ethernet frames carrying TCP/IP data	Maximum-sized Ethernet frames carrying TCP/IP data
Preamble and Start-of-Frame Delimiter	8 bytes	8 bytes	8 bytes
Ethernet Header	14 bytes	14 bytes	14 bytes
TCP/IP Headers	N/A	40 bytes	40 bytes
Payload	46 bytes	6 bytes	1460 bytes
Ethernet Frame-Check-Sequence	4 bytes	4 bytes	4 bytes
Ethernet Inter-packet Gap	12 bytes	12 bytes	12 bytes
Total Packet Size	64 bytes	64 bytes	1518 bytes
Actual Bandwidth Consumed (i.e., packet size plus framing bytes)	84 bytes (672 bits)	84 bytes (672 bits)	1538 bytes (12,304 bits)
Link Speed	1 Gb/s	1 Gb/s	1 Gb/s
Theoretical Maximum Frame Rate	1,488,095 packets per second (approx.)	1,488,095 packets per second (approx.)	81,274 packets per second (approx.)
Theoretical Maximum Throughput	547 Mb/s (approx).	71 Mb/s (approx.)	949 Mb/s (approx).

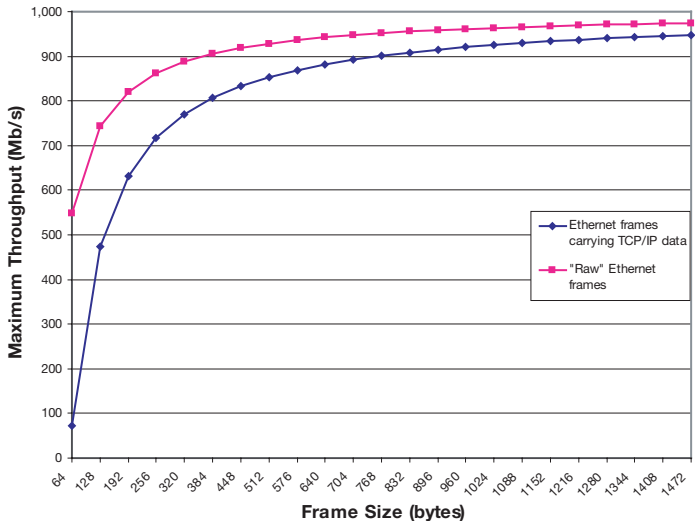

Figure 2. Frame Size vs. Maximum Throughput for 1Gb/s Unidirectional Traffic

Figure 2 illustrates the behaviour described in Table 1. Note that protocol overhead prevents a gigabit device from approaching wire-speed when frame size is smaller than 500 – 600 bytes in length.

3.3 Network Protocol Behavior

The increased frame rate associated with small packet traffic can lead to unexpected protocol behavior in some scenarios. Robust protocols, such as TCP, implement native flow-control mechanisms which attempt to minimize packet losses and network congestion. Simpler protocols, such as UDP, provide no such protection.

When used in conjunction with small Ethernet frames, TCP or similar protocols may demonstrate relatively slow packet rates (well beneath the limits listed in Table 1), but with little or no packet loss. Protocols like UDP may transmit at full-speed but with significant packet losses and network congestion. Applications using these protocols may exhibit similar behavior.

3.4 Per Packet Processing Overhead

Network traffic processing tasks can be classified into two types: Per Byte and Per Packet processing.

Per Byte processing tasks include CRC and Checksum computations. The resources required for these tasks are linear in nature and thus increase as the packet size increases. These tasks have minimal impact on small-packet traffic.

Per Packet processing tasks include all other tasks, i.e., tasks related to the number of packets without regard for payload size. Per Packet tasks include most basic packet processing, such as context switching, buffer allocation, PCI access to the HW registers, lookup and inspection of protocol headers, etc. As the number of packets per second increases, resources required to process these tasks increase significantly.

As an example, consider interrupt processing. If an interrupt is generated for each packet while maintaining the line at maximum utilization, the number of interrupts for unidirectional traffic would be close to 1.5 million interrupts per second! See Figure 3.

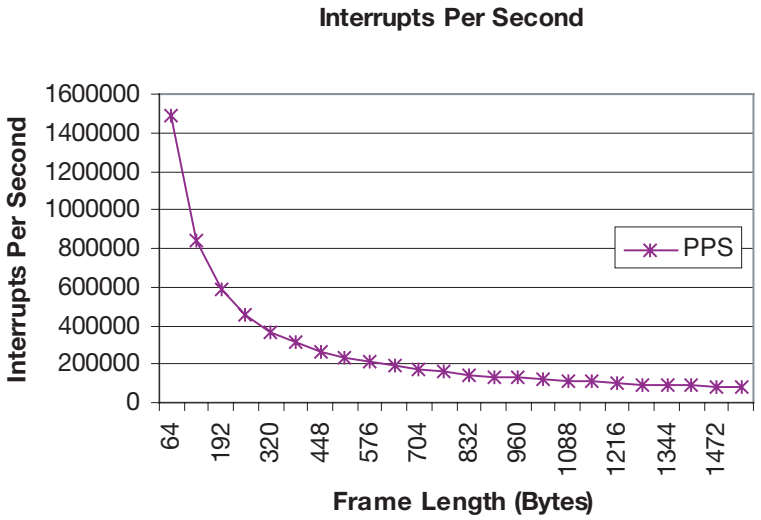

Figure 3. Interrupts Per Second for Different Packet Size

Here, the CPU is constantly trying to service the flood of interrupts, leaving little or no cycles for any other processing! This is referred to as interrupt-live-lock.

A rule of thumb for calculating the cost of sending a packet is 10,000 instructions per packet and 10 instructions per byte (a ratio of 1000-to-1).[1]

3.4.1 Application Related Overhead

Whenever an application calls send() (or receive()) it is actually triggering a number of processing activities. During each send() the operating system must switch context to kernel mode, access the Socket, copy the data to the kernel application buffers, perform protocol related activities, and finally, place the completed packet in the transmit queue. Except for copying data, all processing is proportional to the number of send() calls (rather than to the number of bytes transmitted), and is a major contributor to per-packet overhead.

1. See *Rules of Thumb in Data Engineering,* http://research.microsoft.com/~gray/papers/MS_TR_99_100_Rules_of_Thumb_in_Data_Engineering.pdf.

As an example, a System Call (depending on operating system design and CPU) can require anywhere from a few hundred nanoseconds to several microseconds.[1] System Call overhead can account for as much as 29% of CPU cycles — and Socket related functions, more than 30% (see Figure 4).

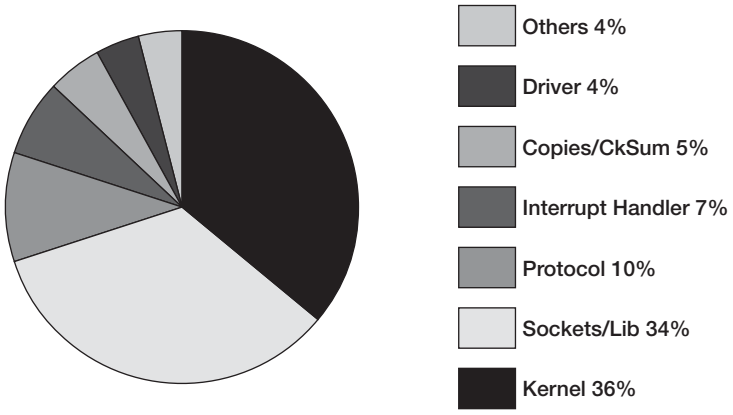

Figure 4. CPU Utilization with Many System Calls and 64-byte Segments (Linux OS)[2]

Compare the utilization above with that shown in Figure 5, where kernel CPU utilization has been reduced to 21% and the Sockets library overhead is zero. Obviously an application designed to send data in many 64-byte segments will consume unnecessary CPU time due to Per-Packet processing overhead.

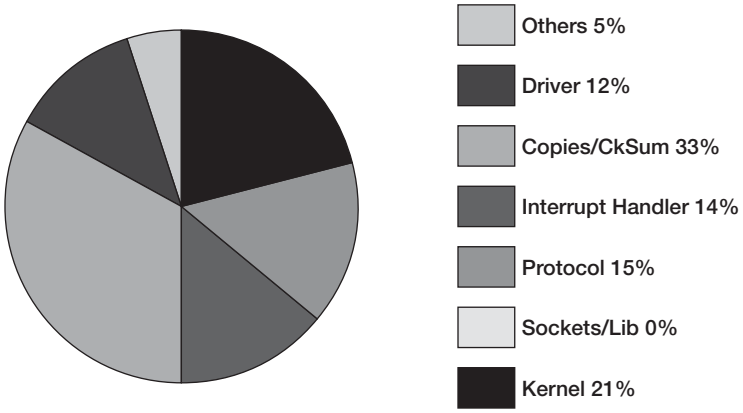

Figure 5. CPU Utilization with Few System Calls and 64-byte Segments (Linux OS)[3]

In some cases, this type of application behavior can lead to poor performance within the protocol. Modern TCP/IP networking stacks employ what is known as the "Nagle" algorithm to mitigate this type of application behavior. However, if the protocol is not TCP/IP, or "Nagle" is not activated during processing, then overall system performance may be degraded significantly.

1. Results from LMBench* tests done by Monta Vista using a 300Mhz Celeron and Linux: Interupt Latency: 98.998% less than 2 µs (worst case, 38 µs); System Call cost: 1 µs; Context Switch cost: 3 µs (0.55 µs with Intel® Pentium III®, according to QNX*)
2. Source: *TCP Performance Re-visited*, Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, 2003 IEEE International Symposium on Performance Analysis of Systems and Software
3. Ibid.

3.4.2 Interrupt Latency Considerations

Interrupt latency is defined as the interval between the time of an interrupt-triggering event and the time that the event is handled. Interrupt latency can adversely impact network protocol behavior and system performance. For example, since packets may accumulate during interrupt latency periods, interrupt latency may determine the number of packets processed per interrupt. Resources must be sufficient to process accumulated packets, otherwise packets may be discarded. Overall performance may deteriorate considerably, if such packet overflow occurs consistently.

3.5 System Bus Limitations

Similar to network overhead issues, PCI/PCI-X bus protocols also impose penalties on small packet traffic. In addition to the actual data transfer, each bus transaction requires extra control cycles which introduce overhead (e.g., arbitration latency, address phases, attribute phases, wait states). This overhead reduces the efficiency of each transaction as well as overall bus performance. For example, placing a 64-byte data packet over PCI-X Bus, requires 8 bus cycles while the overhead of a transaction requires a minimum of 4 cycles (3 to initiate and 1 cycle to terminate). A "worst case" will add up to 16 wait cycles. See Figure 6 for a representation of overhead for several PCI Bus technologies using a 32-byte transfer.

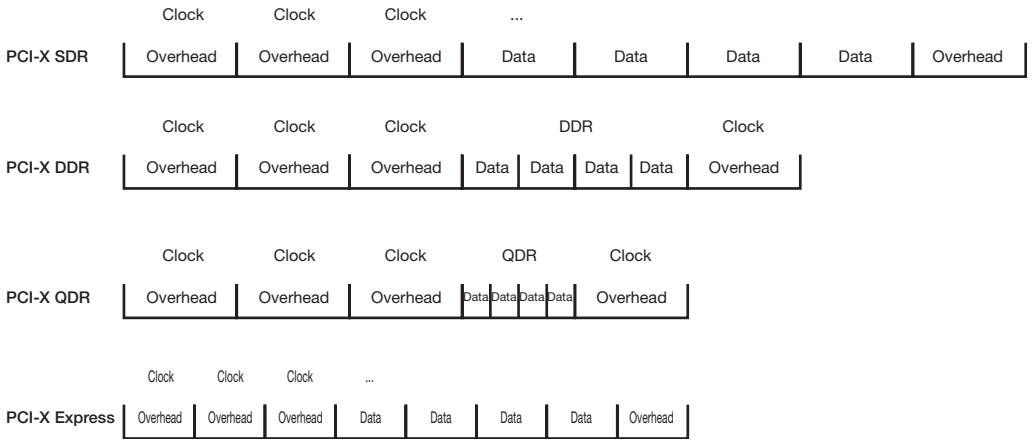

Figure 6. PCI Bus Transaction Overhead

Bus overhead costs become more pronounced with small packet traffic as bus control and wait cycles consume a larger fraction of each bus transaction. Recent increases in the PCI-X data clock rate to Double-Data-Rate (DDR) in PCI-X 266 and Quad-Data-Rate (QDR) in PCI-X 533 only aggravate this situation. DDR and QDR technologies achieve higher data rates by performing more transfers in each data cycle. Although this technique increases the data rate, it does not change the control cycle clock rate, effectively lowering the utilization of the PCI bus for small transactions.

4.0 Optimization Strategies for 8254x and 8255x Ethernet Controllers

This section discusses some general strategies for improving small packet performance of the 8254x-based and 8255x-based controllers. These recommendations may or may not be useful, depending upon the target application.

4.1 Increase Receive Resources Available to the Controller

To boost small packet throughput, software may need to provide the controller with additional receive resources (i.e., receive descriptors, receive buffers).

As show in Table 1, frame rate increases as packet size decreases. This impiles that the Ethernet controller must be able to buffer larger numbers of incoming smaller packets. In addition, the local host will be limited in how many packets per second it can process, therefore, the controller must be capable of absorbing high frame rates on behalf of the local protocol stack. If software does not provide adequate resources for this purpose, the device may starve, and the result will be lost packets and reduced performance.

Increasing the number of receive resources available to the controller will help alleviate potential overrun conditions. Intel® PROSet software (or other configuration tools) may be used to increase these resources. It may also be necessary to increase the operating system resources available to the local protocol stacks.

8254x-based controllers provide statistics that may help indicate resource starvation problems. Large “Missed Packet Count” (MPC) and “Receive No Buffers Count” (RNBC) values may indicate insufficient resources. Adjusting the “Packet Buffer Allocation” (PBA) to increase FIFO space reserved for incoming packets may remedy this issue.

Smililarly, 8255x-based controllers provide “Resource Errors” and “Overrun Errors” statistics that may reveal resource starvation. Both statistics are available by using the “Dump Statistical Counters” command.

Transmit performance is not affected by small packet traffic to the same extent as receive traffic. This asymetry exists because the local host cannot usually overwhelm the Ethernet controller with outgoing traffic. Thus, it is not usually necessary to adjust transmit resources to accommodate smaller packet sizes.

4.2 Optimize Interrupt Moderation

In conjunction with increasing the receive-resources available to the controller, optimizing or eliminating receive-interrupt delays, may reduce the incidence of overrun conditions.

If the controller is configured to delay assertion of receive-interrupts (i.e., interrupt moderation enabled), then the controller will often have multiple frames ready for processing by the time the interrupt occurs. With the higher frame-arrival rates possible when using small packet traffic, the number of packets pending may be significantly larger for small-packet traffic than for mixed-size or large-size packet traffic. "Normal" interrupt delay values may be too large, leading to excessive numbers of packets pending for software at each interrupt, with the final result being packet loss and performance degradation. Reducing or eliminating these interrupt delays allows software to process incoming packets at a faster rate, thereby minimizing the likelihood of overrun.

8254x-based controllers provide a series of mechanisms for adjusting interrupt delays, i.e., the "Receive Packet Delay Timer" (RDTR), "Receive Absolute Delay Timer" (RADV) and the "Interrupt Throttle Timer" (ITR). Large MPC and RNBC values may indicate that these timers require re-tuning.[1]

4.3 Packet Data Polling

In applications requiring high constant frame rates, greater frame rates may be realized (i.e., more frames transmitted/received per second) by polling for completed frames rather than relying on interrupt events. Typically, this requires two changes to the normal interrupt-processing path:

1. The device driver must disable some of the transmit or receive interrupts during initialization, and leave them disabled at runtime.
2. The device driver must use a periodic timer to invoke the interrupt processing logic at regular intervals.

Device drivers for 8254x-based controllers can selectively disable the desired interrupt events using the "Interrupt Mask Clear" (IMC) register. The driver can then scan for completed packets by examining the "DD" and "EOP" bits in the individual transmit and receive descriptors.

Device drivers for 8255x-based controllers can selectively disable the desired interrupt events using the "System Command Block" (SCB) Command Word. The driver can then scan for completed packets by examining the "C" and "OK" bits in the individual transmit command blocks (TCB) and receive frame descriptors (RBD).

Experiments have shown that this type of polling can yield improvements of 20-30% in the number of packets transmitted per second. The optimal polling rate will vary depending on the application and is probably best determined empirically — polling every 30-50µs is a suggested starting point. This method will often affect transmit and receive performance differently.

Alternately, under Linux or Linux-based operating systems, NAPI may be used to implement this polling behavior. Tests using NAPI have demonstrated improvements of 20-25% in receive throughput and packet rate, as well as showing reduced per-packet latency.

1. For more information, see *Interrupt Moderation using the 82540EM, 82545GM and 82546EB Gigabit Ethernet Network Controllers Application Note 450*, Revision 1.0, Intel Corporation.

4.4 Use Simplified Mode (8255x-based controllers only)

To minimize the effects of PCI bus latencies (e.g., bus arbitration, wait-states, etc.), software should use “simplified” mode packet reception.

8255x-based controllers support “simplified” and “flexible” modes of packet reception and transmission. While the “flexible” modes are usually more space and CPU efficient, they are less bus efficient, as the controller must split large data bursts into smaller fragments. Each such fragment requires the controller to re-arbitrate for the bus, transfer some small portion of the data and then relinquish the bus. This constant bus negotiation slows the transfer of packet data between the controller and host memory, thereby slowing the rate at which the device may transmit and receive packets on the network.

To avoid this inefficiency on the bus, software should use the “simplified mode” of packet reception. Where possible, software should also use the simplified method of packet transmission.

Note: This page is intentionally left blank.

5.0 Other Design Recommendations

5.1 Coalesce Small Fragments into Contiguous Buffers

To minimize the effects of PCI bus latencies while transmitting small packets (e.g., bus arbitration, wait-states, etc.), small data fragments should be combined into larger, contiguous blocks. This is most effective when done at the application level. However, a similar effect can be accomplished in the driver by coalescing small packets.

While splitting packet data across multiple small fragments is typically more space and CPU efficient, it is less bus efficient, as the controller must fetch each small fragment individually. Each such fragment requires the controller to re-arbitrate for the bus, transfer some small portion of the packet data and then relinquish the bus. This constant bus negotiation slows packet data transfer between the controller and host memory, slows the rate at which the device can transmit packets on the network, and introduces additional bus contention.

Coalescing fragments into fewer, larger data blocks can boost bus efficiency and transmit performance. For some applications, coalescing packet fragments into two buffers, one for protocol headers, and the other for payload data, may be adequate. It is also possible to coalesce all data fragments into a single contiguous region, which is useful when dealing with very small or minimum-sized frames.

5.2 Kernel vs User Space

Small-Packet processing is usually characterized by a high packet rate, either transmit or receive. As mentioned in Section 3.4.1, a typical application will issue a system call and copy to kernel memory for each packet sent, thereby producing substantial CPU overhead. Depending on the application, if the packet handling is passed down from the user space to the kernel space, avoiding the system call and copy altogether, a significant packet-per-second performance boost can be achieved.

5.3 Hot-Spot Optimizations

High-level languages do not always generate the most efficient code. In circumstances where very high performance is required, low-level coding of ISR and/or Protocol processing can yield substantial performance improvements.

5.4 Buffer Allocation Optimizations

Generic drivers and networking stacks are designed to support all possible packet sizes and traffic patterns. For applications where all system characteristics are controlled, a customized approach can be more effective. A general-purpose OS buffer allocation algorithm is expected to service a wide range of requirements, This flexibility comes at the cost of added complexity and reduced efficiency. However, if, for example, a system requires only a specific packet size, optimizing the buffer allocation scheme can produce improved performance.

5.5 Discarding Packets and Flow Control Schemes

When all resources are exhausted and the system still cannot sustain a high packet rate, flow control and buffer flushing may provide a workaround.

Intelligent use of flow control can maximize the use of bandwidth by minimizing lost processing resources. If the application is concerned with jitter and latency, flow control mechanisms can smooth traffic over time and avoid peak demand which may lead to packet loss. While this is usually done at the protocol level, it can also be accomplished at the Ethernet layer by using 802.3x flow control packets. Both the 8255x family of controllers and 8254x Gigabit Ethernet controllers have hardware support for 802.3x flow control. Empirical data suggests that turning on flow control and setting XON/XOFF thresholds similarly provides an efficient link layer flow-control mechanism.

Finally, in the event that the system must drop packets, the preferred workaround is to drop them early and lower in the stack, rather than wasting resources by processing packets in upper layers. This technique frees CPU cycles and resources for other tasks, however it is heavily dependent on protocol and application characteristics, and any implementation should be very carefully designed.