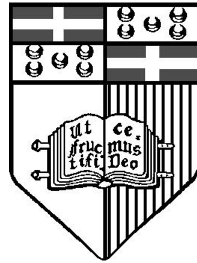
University of Malta
Department of Computer Science and A.I.



Achieving Gigabit Performance on Programmable Ethernet Network Interface Cards

Submitted in partial fulfillment of the requirements for the degree of
B.Sc. I.T. (Hons.)

by

Wallace Wadge

June 2001

Abstract

The shift from Fast Ethernet (100Mbit/s) to Gigabit Ethernet (1000Mbit/s) did not result in the expected ten-fold increase in bandwidth at the application level. In this dissertation we make use of programmable Ethernet network cards running at gigabit speeds to identify present bottlenecks. Furthermore, we investigate the performance of the PCI bus and the throughput "on-the-line" when using different frame sizes. The PCI bus was identified as a major bottleneck. We also propose two new techniques aimed at boosting application performance to near-gigabit levels whilst maintaining full compatibility with existing systems. The proposed techniques were implemented and yielded an 80% point-to-point transfer rate benefit whilst utilizing the existing Ethernet standard framework.

Acknowledgments

I would firstly like to express my gratitude to Dr. Kevin Vella whose patience, perseverance and supervision made this project possible. I am also indebted to Mr. Joseph Cordina for his constant feedback and insights as well as the Computer Science and A.I. Department for providing all the necessary hardware.

On a more personal note I would like to thank my uncle Vincent as well as my parents for their support throughout the years. Lastly, a word of thanks goes out to my girlfriend Antonella, for her constant encouragement throughout the entire project.

Contents

1 Introduction **1**

1.1 Aims and objectives 1

1.2 Structure of document 2

2 Survey **3**

2.1 Conventional Network Cards 3

2.2 Problems with conventional NICs 5

2.3 Recent Work 7

2.3.1 Myrinet 8

2.3.2 GigaPM 9

2.3.3 U-Net 10

2.3.4 Virtual Interface Architecture 10

2.3.5 Bulk transfer protocols 12

2.3.6 Speculative Defragmentation 13

3 Background **14**

3.1 The Peripheral Component Interconnect (PCI) Bus 16

3.1.1 Programmed Input/Output (PIO) 18

3.1.2 Direct Memory Access (DMA) 18

3.2 Ethernet background 19

3.2.1 Frame format 19

3.2.2 Flow Control 21

3.3 The Alteon Tigon NIC 22

3.3.1 Communicating with the Tigon 24
3.3.2 NIC Features 25

4 Analysis of Potential Bottlenecks **27**
4.1 A note about Timings 29
4.2 Theoretical Limits 30
4.2.1 The Jumbo Frames Debate 33
4.3 Determining line speed 36
4.4 PCI Bus 39
4.4.1 Host to NIC 41
4.4.2 NIC to Host 42
4.5 Host transmission 45
4.5.1 Testing transmission from a host source 46
4.5.2 Testing host reception from a NIC source 50
4.5.3 Measuring Host to Host performance 51

5 Packet Fragmentation and Coalescing **56**
5.1 Packet Fragmentation 57
5.1.1 Results 57
5.1.2 The Pitfalls 60
5.1.3 Future Work 61
5.2 Packet Coalescing 62
5.2.1 Results 63
5.3 Host Fragmentation to Host Coalescing 66
5.3.1 Pitfalls 67

6 Conclusion **68**
6.1 Results and achievements 68
6.2 Future work 69
6.3 Final remarks 69

A Data Obtained **70**

A.1 NIC to Host 70

A.2 NIC to Host (with coalescing) 72

A.3 Host to Host (standard firmware) 74

A.4 NIC to Host PCI test 76

A.5 Host to NIC PCI test 78

A.6 Host Split to Host Coalesced 80

A.7 Host Std. to Host Coalesced 82

A.8 Host to NIC 84

A.9 Ethernet Theoretical Limits 86

A.10 Host Split to NIC 88

A.11 NIC to NIC 90

List of Figures

2.1 The VI Interface 11

3.1 A typical PCI configuration 17
3.2 Ethernet Frame Format 20

4.1 Maximum Theoretical Ethernet bandwidth 31
4.2 Maximum Theoretical Ethernet bandwidth (Detail) 31
4.3 Extended (Jumbo) Ethernet Frames vs. Standard Ethernet 33
4.4 Frame sizes analysis 35
4.5 NIC to NIC Unidirectional 37
4.6 NIC to NIC UniDirectional (detail) 38
4.7 NIC to NIC (Theoretical vs. Obtained) 38
4.8 Host to NIC PCI UniDirectional 41
4.9 Host to NIC UniDirectional (detail) 43
4.10 NIC to Host UniDirectional 44
4.11 NIC to Host UniDirectional (detail) 45
4.12 Testing standard firmware (TX) 46
4.13 Host transmitting, NIC receiver benchmarking 47
4.14 Host transmitting, NIC receiver benchmarking (detail) 48
4.15 Host transmitting, NIC receiver benchmarking vs. PCI performance . 48
4.16 Bandwidth difference between Host transmit and DMA performance . 49
4.17 Testing standard firmware (RX) 50
4.18 NIC to Host using standard firmware 51
4.19 NIC to Host using standard firmware (detail) 52

4.20 NIC to Host using standard firmware (detail) vs. PCI Performance . 52
4.21 Host to Host Unidirectional transfer 53
4.22 Host to Host Unidirectional vs PCI performance 54
4.23 Summary of results (1.5K payloads) 55
4.24 Summary of results (9k payloads) 55

5.1 Packet Fragmentation Block diagram 57
5.2 Packet Fragmentation results 59
5.3 Packet Fragmentation results (detail) 59
5.4 Packet Fragmentation vs Host to NIC 60
5.5 Packet Fragmentation vs Host to NIC (detail) 61
5.6 Packet Coalescing 64
5.7 Packet Coalescing vs PCI performance 64
5.8 Bandwidth loss when compared to PCI performance 65
5.9 Host to Host with Packet Coalescing 65
5.10 Host Split to Host Coalescing (block diagram) 66
5.11 Host with Packet Fragmentation to Host with Packet Coalescing (using standard 1.5k packets throughout) 67

List of Tables

4.1 Ethernet Theoretical Maximum Effeciency 32
4.2 Line Speed throughput for different payload sizes 36
4.3 Latency and bandwidth of different Burst length transfers. 39
4.4 Host to NIC UniDirectional (fragment) 42
4.5 NIC to Host (fragment) 43
4.6 Host transmitter, card receiver (fragment) 46
4.7 Host to host (fragment) 53

5.1 Fragmentation results 60

Chapter 1

Introduction

In recent years line throughput has been increased and Gigabit Ethernet has been standardised, yet the expected performance is not being achieved at the application level. Typical Gigabit Ethernet network interface cards, which should be delivering up to 1 billion bits per second, end up providing processes with less than half that rate (around 409 Mbps on Linux-based systems [Weba], less on Windows platforms). Furthermore, these results were achieved using full CPU host loading, which implies that the host system is left with little time for further processing. As bandwidth demands are on the increase, it is evident that this situation can only degenerate further.

1.1 Aims and objectives

In this dissertation, we take a look at each step involved in transferring data from one host machine to another and attempt to locate the main bottlenecks involved in both packet transmission and reception. By making use of special programmable network interface cards, various scenarios, such as transmitting data utilising larger payloads than usual, are simulated. We investigate the following key areas:

NIC to NIC linespeed Firmware reprogramming allows us to accurately verify if current hardware is able to match the expected theoretical results. The effect of

utilising payloads of different lengths is also investigated.

Host to NIC interaction The PCI bus is used whenever data is transmitted or received from the network. One of our aims is to verify the bus's performance by stress testing it with different transaction lengths (from just a few bytes right up to 64k). Both PCI transfer directions are investigated.

Packet processing overheads Another of our aims is to benchmark and potentially reduce the amount of processing involved in transferring data from one host machine to another.

Our major objective is to improve throughput at the application level while maintaining compatibility with existing Ethernet standards. This is achieved by introducing two new techniques, *packet fragmentation* and *packet coalescing*, for the transmitting and receiving side respectively.

1.2 Structure of document

This dissertation is organised as follows. In Chapter 2 we take a look at how a typical NIC works as well as present some recent work aimed at resolving some of the key problem areas. In Chapter 3 we give an overview of all the tools which were utilised in the dissertation while in Chapter 4 we present our benchmarks and analysis of each communication component. Chapter 5 augments the results obtained by describing two new techniques aimed at boosting performance. We conclude in Chapter 6 by highlighting our major results and presenting possible future work.

Chapter 2

Survey

In this chapter we first take a look at how a conventional Network Interface Card (NIC) works in the context of a typical host setup running over the TCP/IP protocol (see [Pos81]). We also take a look at the problems surrounding the usual implementation schemes and summarise recent work aimed at solving some key problem areas. By "conventional network interface cards" we imply the cheap, off-the-shelf, kind commonly found in many offices. These are characterised by having little on-board intelligence and rely mostly on the host to provide many of the necessary functions.

2.1 Conventional Network Cards

When a conventional network card receives a complete unit of data from the network (that is, a packet) and is ready to forward it to the computer's bus, it generates a hardware interrupt. This leads to the network interrupt handler routine being called to encapsulate the packet in some structure. It also enqueues it in the IP queue (or some similar queue, depending on the underlying protocol) and posts a software interrupt. The software interrupt has a higher priority than any user process; therefore, whenever a user process is interrupted by a packet arrival, the protocol processing for that packet occurs before control returns to the running user process. However, software interrupts have a lower priority than hardware interrupts. This

means that the hardware can interrupt the processing of earlier packets.

When packets are transmitted over the network, they may be sent along different routes; thus, at the receiving end, these packets may be received in the wrong order. The IP protocol first reassembles all these fragments and then calls the UDP or TCP's input function (as appropriate). Finally, the packet is queued on the socket queue of the socket that is bound to the packet's destination port.

When a process wishes to send some data across a network, the following steps occur. First, the data is written to a socket (via a system call) to the appropriate buffers. This stage already incurs a performance penalty as a context switch between user and kernel modes occurs. These buffers are then further processed by the underlying protocols such as UDP and TCP and placed on the network driver's interface queue. Packets queued in the interface queue are then removed and transmitted in the context of the network interface's interrupt handler. For stream sockets such as TCP, these buffers are queued in the socket's outgoing socket queue and TCP's output function is called. Depending on the arguments to the send call and the state of the TCP connection, TCP makes a logical copy of all, some or none of the queued buffers; then processes them for transmission and calls IP's output function. Depending on the underlying hardware, the TCP routines may need to split the data up into smaller packets. Many techniques have been proposed to eliminate this copying stage since it has been found to be one of the biggest sources of inefficiency. The resulting packets are then transmitted or queued on the interface queue.

As can be evidenced, the NIC performs a very small part of the work required, most of the effort is left to the host to handle. From a hardware point of view, the conventional system is ideal since, without the need to process packets beyond the Ethernet perspective, there is little circuitry involved. However, there are quite a number of (serious) performance issues, all of which are amplified as speeds increase.

2.2 Problems with conventional NICs

In this section we shall highlight some problems arising out of conventional NIC and OS designs.

Inappropriate resource accounting The reception of a packet results in an interrupt being generated. However, the time taken to handle this packet may not necessarily be taken from the application that is waiting for the data. In other words, it may be perfectly possible for a network user to grab a large chunk of processing time from other users, irrespective of the user's priority. This is unfair since the penalty in performance is at times not attributed to the process waiting for data to be received or transmitted.

Lack of load shedding The need often arises to reject or drop a packet; this would be the case, for example, in the case of a receiver overload. Under a conventional system, packet rejection can only occur after some resources have been consumed.

Lack of traffic separation Incoming traffic designated for one application can lead to a delay and loss of packets for another application.

Interrupt handling The whole system is interrupt driven. Interrupts are quite costly in terms of performance since they result in a mode switch between kernel mode and user mode as well as increased context switching. Packets are typically copied one at a time from the NIC to the host, thus another important measure here is how quickly the data is transferred. A standard Ethernet packet can be up to 1514 bytes long (excluding the 4 byte CRC at the end). Assuming that each I/O bus cycle can transfer 4 or 8 bytes at a time (depending whether the underlying PCI bus is 32 or 64 bit wide) that would mean that it would take between 190 and 379 bus cycles to transfer the data from one place to another. During this time, the running processes in the host are completely blocked.

Eager receiver processing High interrupt priority is given to the capture and storage of packets in main memory while the lowest is given to the applications waiting for the data. Under high network load a host may very well end up using all of its processing time to process incoming packets, only to discard them a short while later. Unfortunately, by the time it discards them, a considerable amount of processing time has already been lost.

This scenario (known as *receiver livelock*) can occur since the earlier receiving processes have a higher priority than later stages. Thus, under heavy load, the consumer process, that is, the process waiting for the data, might find itself continuously interrupted to handle the new incoming packets. Above certain thresholds, more packets are produced than consumed so socket and IP queues start to fill up. Eventually, the queues fill up so all newly received packets end up being discarded – but only after more CPU time has been invested in them. The end result is a host machine doing nothing except discarding incoming packets. This effect can be exploited by denial of service (DoS) attacks whereby a server is flooded by rouge packets starving the host from further work. At the Ethernet level, flow control may aid in controlling this effect.

Checksum calculation overheads Research carried out by Alteon [Webc] indicates that many systems spend as much as 15% of their time just calculating checksums for each transmitted packet.

Buffer alignment Since not all transfers are exactly aligned to 32 or 64-bit boundaries, conventional adaptors usually place the burden on the host to align the buffers prior to data transfer. This is time consuming and further hurts system performance.

Multiple memory copies The running user application must explicitly perform a system call in order to retrieve the data received from the network. This has the effect of copying the internally queued data to the application's address space and finally de-allocating the memory reserved for the copy. Memory to memory copies are done when transferring packets from the NIC to the host, when packets are re-ordered for

the IP and TCP layers and when the payload is finally copied from the TCP protocol layer in kernel space to user space.

All this overhead adds up resulting in CPU utilization rates of up to 90% on the receiver side [GCY].

2.3 Recent Work

Although hardware designers have been placing considerable effort in achieving ever higher peak bandwidths on raw data streams, little concern has been given to the host-side where higher level functions are provided.

Much of the recent research has focused on reducing messaging latency and overheads. The main motivation of these abstractions has always been to reduce or eliminate kernel access in sending and receiving messages to and from the network as well as avoiding memory to memory data copies. With the emergence of new programmable hardware, more and more of the functions usually reserved for the kernel are being transferred either to user-space or else to the hardware itself.

Programmable hardware of the like we see today were not available until a relatively short while ago. Nevertheless, in [BDHR95] we see efforts being directed to reduce kernel access. In their work, the TCP functions were divided into two parts: the demultiplexing functions remained in the kernel while the rest of the protocol moved into user space via a library linked to the application. While a performance gain was noted, the kernel was still heavily involved and thus many bottlenecks remained.

2.3.1 Myrinet

The arrival of ATM brought a new concept to the network world: job offloading. Myrinet was one of the earliest to exploit a system allowing user-level processes to directly access the underlying hardware thus bypassing the operating system kernel [BCF+95]. Myrinet's life was made simpler since its ATM backbone ensured reliability at the hardware level. Myrinet was actually a complete hardware and software solution with the host interface containing many of the features found in the newer programmable NICs including the availability of a DMA engine to move data between the host and the NIC.

A software layer, referred to as the *Myrinet Control Program*, was provided to allow user process to either use the standard TCP/UDP system calls or else use Myrinet's own "streamlined" API. Where it really stands out from earlier work is the way it makes use of external hardware to do some of the necessary processing which is usually left for the host to handle. For example, when it receives a request to transmit a packet of data onto the network, Myrinet invokes the DMA engines to transmit the necessary data and, in operating systems which allow it, off-loads the computation of the IP checksum to the hardware. Myrinet's introduction to the networking world also proved that achieving near gigabit rates was possible on the existing PCI bus.

A common problem with existing network cards is that hardware designers assume a flat view of the system memory and ignore a key feature in most OSs: page swapping. An application is usually not aware that a particular memory page is swapped out and back again since that is done transparently by the OS; however this causes problems for user-level networks. For example when effecting DMA transfers, the DMA engines of the NIC card expects a physical memory address to start transfers. However, the OS may decide to swap the page out during the same time the NIC is busy transferring data. Furthermore, while a process makes use of virtual addressing, the NIC uses the actual physical memory address since it obviously has no knowledge of the OS layer. To circumvent this problem, Myrinet allocates a number of

non-swappable pages at boot-time and expects user processes to manage it via kernel primitives.

This last issue has been further investigated by [TOHI]. The authors argue that if each message transfer involves pin-down[1] and release kernel primitives, message transfer bandwidth decreases since primitives are computationally expensive. The user-level library of PM builds on Myrinet by making the pinned-down memory area reusable by requiring memory to be registered prior to use.

2.3.2 GigaPM

In [STH$^+$99] a mechanism to minimize the latency and maximize bandwidth between the host and the network adapter was designed. As in Myrinet, the authors assume and make use of a programmable network card with it's own dedicated processor. GigaPM builds on the work of PM and Myrinet described above by creating a system of *message descriptors*. Unlike earlier systems, these descriptors are stored and manipulated directly by the NIC (the *Essential Communications PCI Gigabit Ethernet NIC* in their case). GigaPM makes use of PM's pin-down techniques and Myrinet's design philosophy but shifts more of the necessary work on the NIC by sending messages in the form of descriptors while letting the NIC poll for any changes. This has the effect of not involving the kernel at any stage. GigaPM also defines a new type in order to let TCP/IP and other protocols co-exist with it. Interestingly, the *Acenic Tigon NIC* has been designed from the ground up to support this method of communication (more details on how the Acenic Tigon is designed will be given in Chapter 3). Unlike Myrinet, GigaPM does not assume reliable message transfer at the hardware level so issues such as to where error detection and correction occurs remain.

[1]The communication buffer area is reserved as a special virtual address area whose physical area is never paged out. This is known as pinned down cache.

2.3.3 U-Net

As several research papers have pointed out, eliminating the kernel from the send and receive paths requires that message multiplexing and demultiplexing has to be performed somewhere else in hardware or in software in order to enforce protection boundaries. U-Net [vEBBV95] tries to consolidate much of the work done by Myrinet, PM, and GigaPE as well as move the multiplexing and demultiplexing routines directly onto the network interface. It also moves all buffer management and protocol processing to user-level. This gives the illusion that each process owns the network interface. U-Net not only provides multiplexing of the network interface among all processes accessing the network, but also enforces protection boundaries and resource limits. A user-level process has all the control of the network card it can get with the exception of the source and destination addresses as well as buffer resources. Without this protection, a process may easily disrupt the whole host system by specifying a destination buffer address which was not reserved for its use. Unlike normal processes, the underlying hardware is not kept in check by any protection boundaries and is free to overwrite any part of the hosts' memory. U-Net is composed of three main building blocks – segments of memory to store message data, message queues to hold descriptors for messages that have been received or that are to be sent and endpoints which are used for control. As in GigaPM, when a process needs to send a packet to the network, it adds a message descriptor to the appropriate queue and waits for the NIC to eventually pick up the data and transmit it.

2.3.4 Virtual Interface Architecture

Hindered by all the different proprietary solutions, a new standard is being promoted. Currently supported by more than a hundred industry organizations (including Compaq, Intel and Microsoft), this standard, known as the *Virtual Interface Architecture* or VIA [DRM+98], broadly defines the way a process and the network interface hardware should communicate. The VIA is closely related to the U-Net design, borrowing

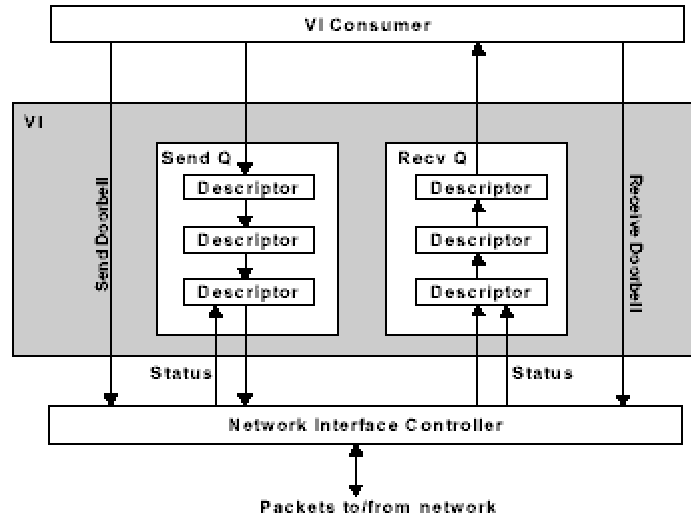


Figure 2.1: The VI Interface

many of its design features by providing each process a protected, directly accessible interface to the network hardware. The tasks of multiplexing and de-multiplexing and data scheduling (normally performed by the OS kernel and driver) are also moved onto the hardware itself. The VIA standard does not assume hardware reliability (that is, it can be implemented on both ATM, Ethernet and other platforms). The standard only expects NICs to support unreliable delivery, that is, a send request will be processed *exactly* once and corrupted transfers are detected at the receiving end (but retransmission of lost packets is not required). In other words, all that is required is that if the NIC detects an invalid packet, it has to mark the frame as bad by switching on the appropriate status bits.

The VI Architecture is comprised of four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers (see Figure 2.1). The VI Provider is composed of a physical network adapter and a software kernel agent. The VI Consumer is generally composed of an application program and an operating system communication facility. The virtual interface is the mechanism that allows the VI consumer to directly perform the data transfer operations. All communication takes place in the context of completion queues with doorbells used to signal each

other. This is the tried-and-tested model used by U-Net and the *Acenic Tigon NIC* whereby processes add descriptors to a queue and the NIC picks them up and process them.

The VIA specification requires that processes identify the memory to be used for data transfer by registering it. This memory registration scheme allows processes to re-use the assigned memory by locking down the memory pages to avoid the OS from paging the memory segment out to disk, while avoiding the performance overheads of repetitive locking and unlocking.

The VIA specification is intended to be implemented in an environment where multiple processes make use of one or more NICs installed in a system and thus a good part of the specification describes methods to safely run multiple virtual interfaces together.

In this dissertation we are concerned with getting the highest point-to-point data transfer rate between two hosts and we always assume a single interface card used exclusively by a single process.

2.3.5 Bulk transfer protocols

Despite the attempts by the above models to achieve zero-copy handling, a last defragmenting step was still necessary to separate the headers from the data. Some researchers took the assumption that when receiving multiple packets, the destination host can optimistically expect that the next packet to be received is the next packet in the transfer [CZ89]. They thus devised special protocols coupled with page remapping techniques (to avoid memory copies). However, while the TCP/IP and blast protocols can co-exist, they do not interact; in other words, it requires a user-process to be aware of such facilities and make use of them. Not surprisingly, though blast techniques are actually more than a decade old, none of them have made it to the mainstream API.

2.3.6 Speculative Defragmentation

On a more interesting note, [CKS00] proposed a system whereby the driver assumes that packets will arrive in order and generates no less than six DMA descriptors per packet to separate the headers from the payloads immediately (since the system always works within a memory page of data, which is around 4K). Though, the technique on its own gives very little performance benefit, when coupled with page remapping into user space, they were able to show that a large performance benefit could be obtained as the last defragmenting copy was avoided. However, the method has three main drawbacks. Firstly, since the DMA descriptors at the receving end are pre-fabricated, a non-ordered arrival of packets results in some work being done to reorder the payloads. The authors argue that the best case scenario, that is, when all packets arrive in order, is the most common scenario. Furthermore they show that when this occurs the performance loss when this occurs is not that high. This may be true on a low number of switches but may not necessarily be true on more highly connected clusters. On a worse note, the fact that the driver is doing additional work on the host side can only mean that the CPU utilization rate increases. No figures were available to contrast the standard protocol with their methods. Furthermore, DMA startup costs have been shown to be expensive and the system uses no less than six DMA descriptors of which half of them are only a few bytes long (more specificially, the 14 byte Ethernet header) while the rest transfer a maximum of 1500 bytes each. In Chapter 4 we show that DMA performance is highly dependant on the number of bytes transferred.

Chapter 3

Background

The shift from text-based services, such as e-mail, to bandwidth intensive applications, such as real-time audio and video, as well as an increased reliance on distributed databases scattered around a network has resulted in ever increasing demands being placed upon existing network systems. From the simple requirements of a few years ago, an increasing number of enterprises are now demanding high-volume, low-latency transmission of terabytes of data, distributed over a multitude of platforms and accessed by potentially thousands of users at one time. From its humble beginnings over 30 years ago and facing stiff competition from other cabling technologies, Ethernet has survived to become the most popular networking technology for local area networks today. Ethernet has survived for so long chiefly due to its very low cost, reliability and ease of deployment. Originally conceived as a small internal project to share a printer in a small office (a novel idea at the time), Ethernet has grown to become the de-facto standard for *cheaply* connecting two systems together. Initially, Ethernet systems could provide up to 10 Mbps of raw line speed which was more than enough for the demands at the time (this system was known as 10BASE-T). As demands grew, 10BASE-T gave way to Fast Ethernet (100BASE-T), which could, theoretically, transmit at 100 Mbps. Fast Ethernet used the same cabling as 10BASE-T, while everything else – the packet format and length, error control, and management information – remained the same. Its scalability ensured its quick adoption. Initially, Fast Ethernet was used for backbones while existing 10BASE-T systems were

retained for the workstations. As demands grew, more and more desktop systems required this high level of throughput hence triggering the need for yet another bandwidth upgrade, at least at the backbone level. Thus, once again, the requirement for higher throughput led to the development of what we now call Gigabit Ethernet. Work is already underway to standardise the next evolutionary step — 10-Gigabit Ethernet, which should push Gigabit Ethernet's theoretical 1 billion bits per second to even higher levels.

Until recently, it could be assumed that network speeds would be far slower than CPU and bus speeds. However, LAN network speeds have been increasing at a faster rate than CPU and bus speeds. While great efforts have been made to ensure that data could physically move from one end-point to another at the fastest possible rate, much less emphasis has been placed at the receiving ends, the host systems processing the data. It has always been assumed that the problems already evident when 10BASE-T (10 Mbps) Ethernet first appeared on the market would diminish as host processors got faster. This, as we shall be seeing, was not to be the case with the end result being a system failing to meet user expectations. Ethernet operates as a "best-effort" data delivery system. To keep the complexity and cost of a LAN to a reasonable level, no guarantee of reliable data delivery is made. While the bit error rate of a LAN channel is carefully engineered to produce a system that normally delivers data extremely well, errors can still occur. A burst of electrical noise may occur somewhere in a cabling system, for example, corrupting the data in a frame and causing it to be dropped. Or a LAN channel may become overloaded for some period of time, which in the case of Ethernet can cause 16 collisions to occur on a transmission attempt, leading to a dropped frame, which is why higher protocol layers of network software are designed to recover from errors. It is up to the high-level protocol that is sending data over the network to make sure that the data is correctly received at the destination computer. High-level network protocols can do this by establishing a reliable data transport service using sequence numbers and acknowledgment mechanisms in the packets that they send over the LAN (for example, via the use of the TCP protocol). This transport service is normally performed by the host system

itself, which implies that sending information from one endpoint to another requires considerable host assistance.

Reliable data delivery (and consequently, host loading) is not the only issue present here. Existing hardware designs may lead to bottlenecks further up in the communication chain. For example, interrupt handling may work well for lightly loaded systems, but under a heavy load, a system might find itself completely flooded by such interrupt requests.

In this chapter we discuss the tools available to us to aid in understanding how our results were produced. We shall be giving a brief overview of the PCI bus, the Ethernet frame format and our main tool, the Acenic Tigon NIC.

3.1 The Peripheral Component Interconnect (PCI) Bus

The *Acenic Tigon NIC* uses the PCI bus for all communication between the host and itself, that is, the PCI bus is used to transfer data from the hosts' memory to the NIC and vice-versa. First released in 1992, the PCI bus was designed to solve many of the problems with older architectures, while at the same time delivering a substantial increase in processing speed.

The PCI specificication [Gro] originated as a signal-level hardware specification and thus board vendors are free to devise their own implementations. As a result, actual implementations of the PCI standard vary widely. A test of our systems showed a maximum bandwidth difference of more than 200Mbits/s between different implementations of the specification. This is probably due to the manufacturers' rush to market, whereby some ISA designs were ported to the PCI bus in an effort to min-

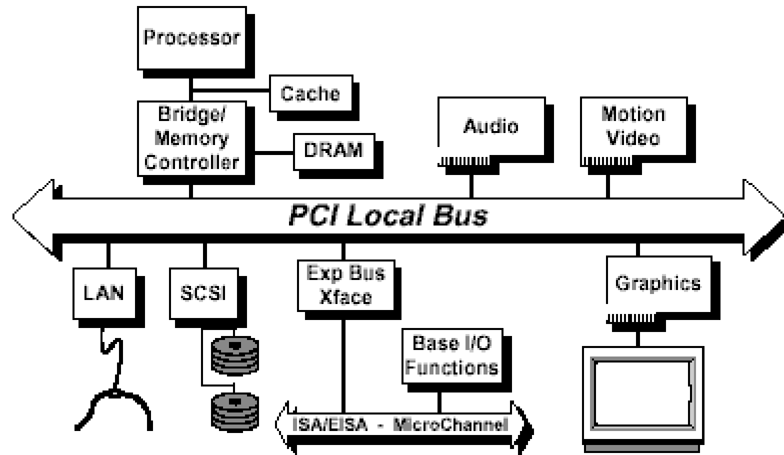


Figure 3.1: A typical PCI configuration

imise costs.

The original PCI specification was defined to run at 33Mhz with 32 bits transferred per cycle. Provisions have however been made to run the bus at 66Mhz and up to 64 bits transferred per cycle. Such systems are already making their appearance on high-end servers. Nevertheless, the 33Mhz/32-bit versions are by far the most commonly available chipsets available and we therefore decided to implement and test our dissertation on such a configuration.

Given that our chipsets run at 33Mhz with 32 bits transferred each time (4 bytes), then the maximum bandwidth from the host to the NIC (or vice versa) we can ever hope to achieve is :

$$(33 \times 10^6) cycles/s \times \frac{32bits}{8} = 132000000 \ bytes = 125.89MB/s.$$

Of course this implies the best case scenario with the chipsets complying perfectly to the specification (by achieving the expected transfer rates) with just one interface card utilizing the bus during that time. Later on we shall be testing the peak performance on our testbed systems and compare it to the maximum theoretical throughput

possible.

The PCI component and add-in card interface is processor independent. This means that the PCI Local Bus can be optimized for I/O functions since it does not require any processing on the host CPU. This is important for it means that we can move data between the memory subsystem and the interface cards without involving the host CPU at all. There are, however, two supported modes of transferring information between two endpoints, PIO and DMA, both of which we describe below.

3.1.1 Programmed Input/Output (PIO)

In PIO, the host processor is set up to move every byte by explicit load and store instructions to memory or I/O mapped addresses. In other words, an application or the hardware itself invokes the processor to execute a loop where a byte, word or double word is transferred each time, according to the instructions used. From the hardware point of view, this is a very simple implementation for it does not require a great deal of hardware complexity, however PIO transfers suffer from two major drawbacks. Firstly, the maximum bandwidth possibly obtained is lower than the DMA counterpart. This occurs because PIO requires the CPU to first check for the availability of the data, then read the data, and then write the data thus reducing the maximum possible bandwidth (trivially, the possible bandwidth speed is dependant on the CPU speed). Secondly, since PIO uses the host CPU this implies that, for the duration of the transfer, running processes will be starved of all CPU time. To its credit, PIO transfers result in lower latencies than other systems such as DMA or Bus Master DMA.

3.1.2 Direct Memory Access (DMA)

DMA is another way to transfer data to and from interface cards or memory locations. There are two ways a DMA can operate:

CPU-based DMA A DMA controller (often on a PC's motherboard) seizes the bus periodically (for example, once for every 16-bit transfer) to read data from the adapter, then seizes it again to write it to memory (therefore requiring two bus cycles per transfer). This reduces costs since individual interface cards need not implement DMA logic onto the boards themselves. However, since two cycles are needed, performance tends to drop.

Bus Master DMA Overall, this is a faster data transfer technique than standard CPU-based DMA. The peripheral writes the data from its memory directly to the PC's memory in one bus cycle (reducing the load on the bus), rather than the usual two-step process. This technique requires interfacing with the memory controller requiring some additional circuitry. Often, a NIC adapter will do its transfer as the data is received so no, or little, on-board adapter memory is required, hence reducing costs. Bus master DMA is faster than standard DMA, since the CPU does not even need to load the DMA registers (for example, with the source and destination addresses) to set up each transfer.

For smaller transfers, the time taken to set up all the DMA controller registers may be longer than the time saved by using DMA, so in this case using PIO may be faster than DMA.

The most important consideration to keep in mind here is the fact that the setup time required to start a DMA transfer is constant irrespective of the number of bytes transferred.

3.2 Ethernet background

3.2.1 Frame format

One of the reasons why Ethernet has caught on so quickly is due to its simplicity in design. The frame can be broadly described as having a *header*, a *payload* and a *trailer*.

The header describes the length of the payload as well as the source and destination addresses. The actual data to be used by the receiver is called the payload and may theoretically be up to 64k bytes long, however the Ethernet standard dictates that the payload length be between 46 and 1500 bytes long. At the end of the payload, a calculated CRC (on the header and payload or on the payload only) is added. This is used by the receiver at the other end to determine whether the frame's integrity was maintained or not. The *Alteon Tigon NIC* used in this dissertation can be programmed to automatically append such a value to all outgoing frames automatically. The preamble and postamble are of no relevance to us for they are used by the hardware to determine the start and end of a frame (they are just known simple patterns for easy hardware detection).

Bits	64	48	48	16		32	8
Content	Preamble Address	Destination Address	Source Field	Type/ payload	Data	CRC	Postamble

Figure 3.2: Ethernet Frame Format

The most important observation to be drawn here is that the Ethernet header has a *constant* size, that is, irrespective of the payload's length, the header will always be exactly 14 bytes long. This means that smaller packets have more overhead than larger ones, since the ratio of overhead is proportionally higher to the amount of data passed. As a result, the data throughput on the line decreases. However, small packets have a reduced chance of retransmission, better response time, and are less likely to contain errors. On the other hand, larger packet sizes have a better ratio of overhead to data, which increases throughput; yet, buffer and transmission delays and the resulting retransmissions can act to degrade throughput. We can summarise this as follows:

As the payload length increases, the overall overhead costs decrease, at the expense of larger latencies and an increased chance of packet corruption.

3.2.2 Flow Control

When a device is receiving information at a fast rate it may very well end up unable to cope with the influx of data. This can happen, for example, when the NIC is unable to shuttle the received frames to the host fast enough, as in the case of a slow bus. Eventually, when the receiver's buffers fill up, the NIC would have no option but to drop incoming frames leading to the higher level protocols to request retransmission. To avoid or minimise this scenario, a way was devised to signal the sender to pause transmission for a short while.

Half Duplex Flow Control A non-standard but popular scheme called Back-pressure is used in half-duplex links. In this scheme, when a half duplex device is not able to handle the amount of data received from an end station, it pretends that a collision has occurred. Thus all devices on the shared LAN would have to backoff, and then try to retransmit over and over again until the receiver was ready to accept the new frames.

Full Duplex Flow Control Since there are no collisions in full-duplex, a different scheme for implementing flow control on a full duplex link was necessary. The solution was to make use of special *control frames*, PAUSE and RESUME, to signal the sender to stop sending for a while or resume retransmission respectively.

If a NIC waits until its buffers are full prior to sending flow control frames it would be perfectly possible for frames to be lost. This can occur because by the time the control frame reaches the sender, additional frames may have been sent already. The solution is to define threshold levels as to when a signal should be sent out. Of course

this technique does not guarantee that no frames would be lost; this would be the case, for example, where the flow control packets never make it to their destination. The Tigon does not automatically send out flow control packets since that is a job left for the firmware to handle by setting the appropriate pause/resume threshold levels.

3.3 The Alteon Tigon NIC

In this section we describe the Alteon Networks Gigabit Ethernet NIC (or Tigon for short), which is the NIC we used for our implementation.

The NIC is built to the PCI specification supporting both bus widths (32 bit and 64 bit) as well as both bus speeds (33Mhz and 66Mhz). The card detects the appropriate bus type automatically but in our case, this is always the 33Mhz/32-bit variety.

The Tigon does not run an in-built fixed programme like other ROM based NICs. Instead, the software required to control the Tigon is downloaded to the adapter during the initialization process (this is done without the need of some special firmware loading utility and without the need of PROM swapping). In order not to confuse it with the host driver, this software is referred to as the *firmware*. At each debugging cycle we therefore download the latest revision of the firmware onto the Tigon and instruct the NIC to execute it.

Alteon Networks used to provide the source code of the distributed firmware for the NIC in an open source format (at time of writing this is no longer the case since the Tigon has been sold to 3COM). Since we will be making changes to the firmware, in this document we refer to the supplied firmware as the *standard firmware* to distinguish it from ours. It is up to the firmware to control most of the hardware, including two independent DMA engines (for transmitting and receiving), MAC hardware and even controlling the traffic and link LEDs. This alone makes the Tigon extremely flexible as regards to what it can do for it can always be "upgraded" by modifying its firmware.

The Tigon contains two on-board RISC-based processors. The processors can be used for any function including parsing buffer descriptors and controlling the DMA hardware.

The Tigon also features two, completely independent DMA channels, one used exclusively for host memory reads while the other is reserved for host memory writes. The maximum number of bytes the hardware can transfer in a single DMA operation is 64kb. The Tigon NIC available provides 1Mb of memory which must also include the firmware and various data structures, leaving approximately 800kb of buffer space for transmission and reception.

External communication is via the Gigabit Ethernet MAC on a fibre channel (though at time of writing copper versions were also in the pipeline). The MAC is able to support both the existing 10/100 modes as well as the new 1000Mbit standard.

Interrupts are never generated automatically from the hardware. Again, firmware has complete control on when to generate an interrupt to the host, if at all desired. The firmware may opt to coalesce interrupts until a number of packets are received and then generate a single interrupt. However, in our firmware we completely mask out interrupt generation relying on polling instead. The embedded processors follow the event model rather than an interrupt model. This implies that the host is never interrupted from the task currently running. The firmware is helped by considerable hardware assistance in running this model via the provision of special hardware instructions.

The Tigon also contains a programmable timer which can be controlled by the firmware to provide time-outs, interrupts and any other function in which it not be suitable to rely on software spin loops. In our implementation we also make use of this timer in order to get time values for some of our benchmarks. This is necessary since if we were to rely on the host to keep track of the time elapsed, the delay in waiting for the completion signal from the NIC would lead to inaccurate results.

3.3.1 Communicating with the Tigon

The primary way the host and the NIC communicate is via the use of a series of shared[1] rings. There are two indices (producer and consumer) that control the operation of each ring. The producer adds the elements to the ring and increments the producer index while the consumer removes elements from the ring and increments the consumer index. When the producer and consumer indices are equal, the ring is empty; otherwise if the producer is one behind the consumer, the ring is full.

Whether the NIC is the consumer or the producer depends on the operation being carried out. For example, when the host wishes to send a frame onto the network, the following steps occur:

1. The host creates a frame in host memory
2. The host creates a buffer descriptor in the TX ring that describes the frame
3. The host updates the Send Producer and writes it into a mailbox (a doorbell whose value may be significant) in a shared memory region.
4. The mailbox triggers an event in the NIC. The producer is thus checked and the NIC starts working on the ring, DMA-ing the frames from the host to the NIC and out on the network.

In this case the host performs the producer tasks while the NIC "consumes" the new descriptors. For the receiving side, a similar ring mechanism is used. The NIC also performs some other minor tasks such as informing the host that the frame has been consumed and so on but these have been omitted from the above list for clarity. There are two other kinds of ring available: the *command ring* and the *event ring*. The command ring is used by the host to instruct the NIC to perform some specific task such as instructing it to halt either of its internal processors. The event ring on the other hand is used by the NIC to inform the host of non-transmission-related

[1] Hardware ensures that host memory updates are reflected in the NIC and vice-versa

events such as to inform the host that the firmware is up and running. The NIC is also responsible for updating statistics such as the number of frames transmitted, received and so on. However, in order to avoid our benchmarks from being affected by statistics transfers, we switch the feature off. At a lower level, memory can be shared between the NIC and the host, with the hardware reflecting memory writes transparently. This is also the way the firmware is downloaded upon initialization.

3.3.2 NIC Features

The Tigon also has a number of useful features which we exploit to improve performance:

Mailboxes are a common technique to facilitate communication between host processors and adapters. Any value written to the mailbox locations has the effect of triggering events on the embedded processor. This is useful, for example, for a host to inform the NIC about changes in the consumer and producer indices.

Buffer Alignment Since not all DMA transfers are aligned to 32 or 64 bit bounteries, many adapters place the burden onto the host to align the buffers prior to data transfer. The Tigon includes logic to permit data to be transferred from *any* host address to *any* byte offset of the Tigon's internal memory.

Checksum Offloading The NIC can do complete hardware-based checksum calculation for both transmitted and received frames. This checksum offload feature also allows full checksum calculation for IP, TCP and UDP packets. According to Alteon Networks [Webb], performing TCP/IP checksum in adapter hardware instead of using server CPU cycles saves around 2.3 server CPU cycles for every byte of data.

The DMA engines can also be configured to calculate a TCP-style checksum while effecting transfers and store the result. Unfortunately, due to inherent hardware bugs, the second internal processor is programmed to take over this function normally

performed by the hardware. Throughout our implementation therefore, we always assume the availability of one internal processor rather than two.

Chapter 4

Analysis of Potential Bottlenecks

Prior to discussing each component in turn, we summarise the entire communication cycle so as to place each key area in context.

A normal application usually makes use of operating system calls to handle both input and output routines. Depending on the underlying protocol, the operating system's communication routines would then wrap the data into Ethernet packets and transfer each one to the NIC for transmission. At the receiving end the NIC would, upon reception of a packet, generate an interrupt to signal the OS to transfer the packet from its buffers to main memory. The OS finally unwraps the packet's header and trailer and presents it to the application as a stream of data.

In our case we shall completely bypass the operating system and operate directly from user-space relying on the OS merely to set the link up. Instead of interrupts, we shall be using the U-Net model [vEBBV95] by writing descriptors in specific places in memory and rely on doorbells (mailboxes) to signal the presence of new descriptors. We will not be dealing with higher level protocols such as TCP since most of the techniques described hereunder will equally apply to it. We also make use of a special kernel level driver named "consequetive" (*sic*) to pin-down memory at boot-time forcing the OS to exclude the area from its virtual memory mechanisms. The dissertation comprises modifying the existing firmware to perform some benchmarks as well as provide additional functionality.

There are actually four distinct programmes we require, each performing a specific task. These are:

Host software on the sender side In tests that require it, the host software grabs a chunk of the pinned-down memory and constructs the packets directly in it. Unlike an application making use of the OS routines, the host is also responsible for creating the packets in the right Ethernet format. In other words, to send a packet the application has to first allocate a chunk of *contiguous* memory. The packet header is written with the appropriate fields and a descriptor for the packet is constructed. The final step is to ring the NIC's doorbell by writing the appropriate producer index in its mailbox. The sender is rarely involved in benchmarking since there is no accurate way of determining when the packet has been physically sent out onto the network by the NIC.

NIC firmware on the sender side When the host adds a send descriptor and rings the doorbell, the NIC receives an event which eventually calls up the appropriate function in the firmware. The firmware is then responsible for starting the DMA engines to transfer the packet from host memory to a specific location onto the on-board memory and enqueue the local address and length onto the MAC queue. In some of our tests we modify the standard firmware to provide additional functionality.

NIC firmware on the receiver side When the NIC receives a packet, it first checks for available buffer space on the on-board memory. If memory is available, the NIC looks into the receive ring in order to obtain the next buffer descriptor. This buffer descriptor is created by the host to indicate to the NIC where, in host memory, it may place the incoming frames. The NIC then signals to the host that it may process another frame and the cycle starts again. In some of our tests, we extend the standard firmware to perform some additional tasks.

Host software on the receiver side Like the sender, all packet handling is done by the user-process without involving the OS at any stage with the exception of the initial set-up. This software also contains the necessary benchmarking routines. Receiving data from the NIC involves the host allocating enough contiguous space in host memory, sending a descriptor to the NIC to inform it of the location and length of the available memory chunk and finally getting back information from the NIC in another ring.

We can thus identify three potential bottlenecks:

- The transmission of a packet from the host sender onto the NIC,
- The physical transmission of the packet from one NIC to another, and
- The copying of the packet from the NIC back onto the host.

As in every pipeline, the slowest link in the chain will define the maximum throughput which can be obtained. We shall be investigating each area in turn.

4.1 A note about Timings

Most of the results presented hereunder are expressed as a measure of the bandwidth obtained. To time the results we have used one of the following two methods:

Timing directly from the NIC In this setup, the NIC's on-board timer is used to provide the necessary timing functionality by simply comparing the initial timer value to the value at the end of the test. This timer has the advantage of being unaffected by any external events such as interrupts or power management features. This timer has a 1μs resolution, but fluctuates slightly with changes in the NIC's on-board temperature.

Host timing Whenever the host is set to receive packets, the host measures the time elapsed by sampling the number of CPU clock cycles via the Time Stamp Counter

(TSC) register. This register keeps track of the number of CPU cycles since the last boot and has a nanosecond resolution. Since the interpretation of this value depends on the CPU's speed, an initial test is performed in order to determine the number of CPU cyles per second. Unfortunately, this has to be done via the use of the hardware timer chip which has a finite resolution, and the smallest value by which it can be incremented does not usually divide equally. A sample measurement showed that this timer varied by no more than ± 0.15% though results show that the bias tends towards the -0.15%.

4.2 Theoretical Limits

The first step in any scientific investigation is to determine the expected results since this would serve as a baseline for all our other results. We therefore first calculate the maximum bandwidth we can ever hope to attain. The Ethernet frame format shown in Figure 3.2 reveals that, for any payload length, the total frame length will be equal to the payload length added to the header, trailer and preamble lengths. An 11-byte long interpacket gap is also "transmitted" between one frame and the next. Our true frame therefore contains the following components:

Header	Payload Length	CRC	Preamble	IPG
14	46-1500	4	9	11

This implies that, in order to obtain the true frame length, we have to add 38 bytes to whichever payload length we choose.

An analysis of the calculations reveals that our 1 Gbit/sec limit can theoretically be reached without the need of very large packets. This is important because Ethernet error detection techniques put an upper limit on frame size. When a frame is transmitted, the sender computes a 4-byte value derived mathematically from all the other bits in the frame. At the receiving end, this number is computed again and the packet is discarded if the computed value does not tally with the received CRC value.

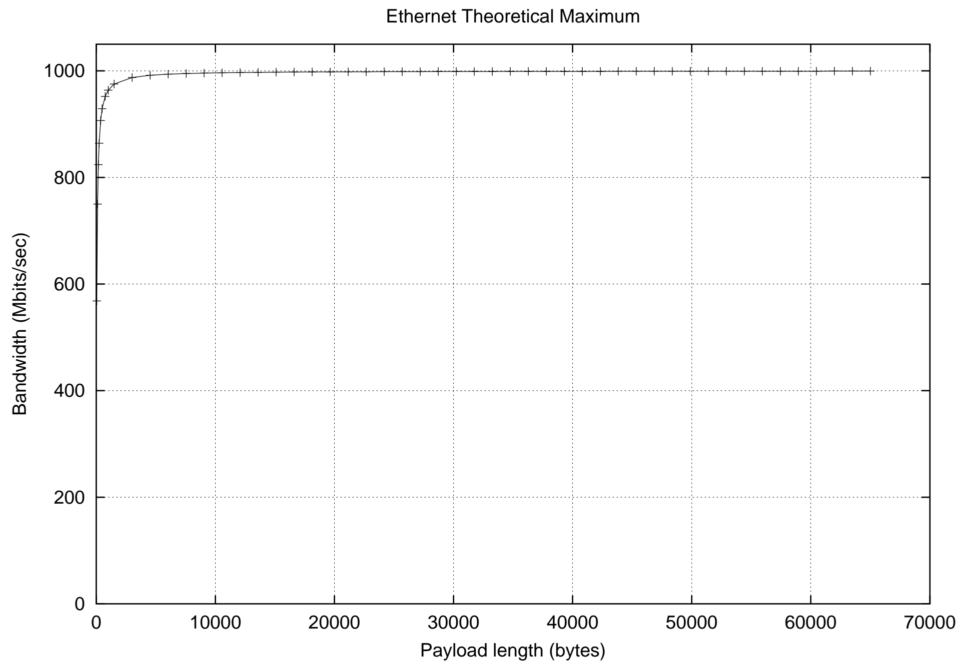


Figure 4.1: Maximum Theoretical Ethernet bandwidth

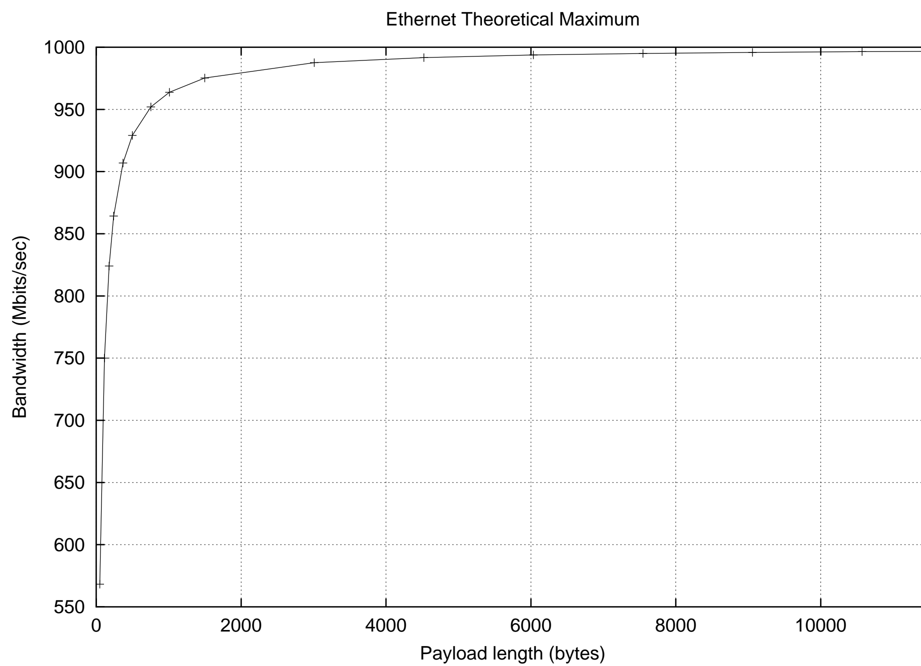


Figure 4.2: Maximum Theoretical Ethernet bandwidth (Detail)

This error checking detects bit errors with a very high probability, but as the frame size increases, the probability of undetected errors per frame may increase. Due to the nature of the CRC algorithm, the probability of undetected errors is the same for frame sizes between 376 and 11,455 bytes. Thus to maintain the same bit error rate accuracy as standard Ethernet, frames should ideally not exceed 11455 bytes. Taking a closer look, one can see that, with smaller payloads, the maximum bandwidth drops dramatically, and this can be problematic since some applications make heavy use of them.

There are two more important observations to be derived. Firstly, the maximum bandwidth possible using standard 1500-byte Ethernet packets can never exceed 975.26 Mbits/sec. Secondly, at the cost of increased latency, the maximum bandwidth which may be obtained by larger packets is little more than the bandwidth obtained for standard packets. In Table 4.1 we present some of our calculated values. Note that we quote the value for a 1498-byte payload rather than the 1500-byte maximum. This is intentional for comparison with some of the results we will be presenting later on.

True Frame size	Real Payload	Theoretical Max Efficiency
792	754	952.0202
1048	1010	963.7405
1536	1498	975.2604
3048	3010	987.5328
4560	4522	991.6667
6072	6034	993.7418
7584	7546	994.9895
9096	9058	995.8223

Table 4.1: Ethernet Theoretical Maximum Effeciency

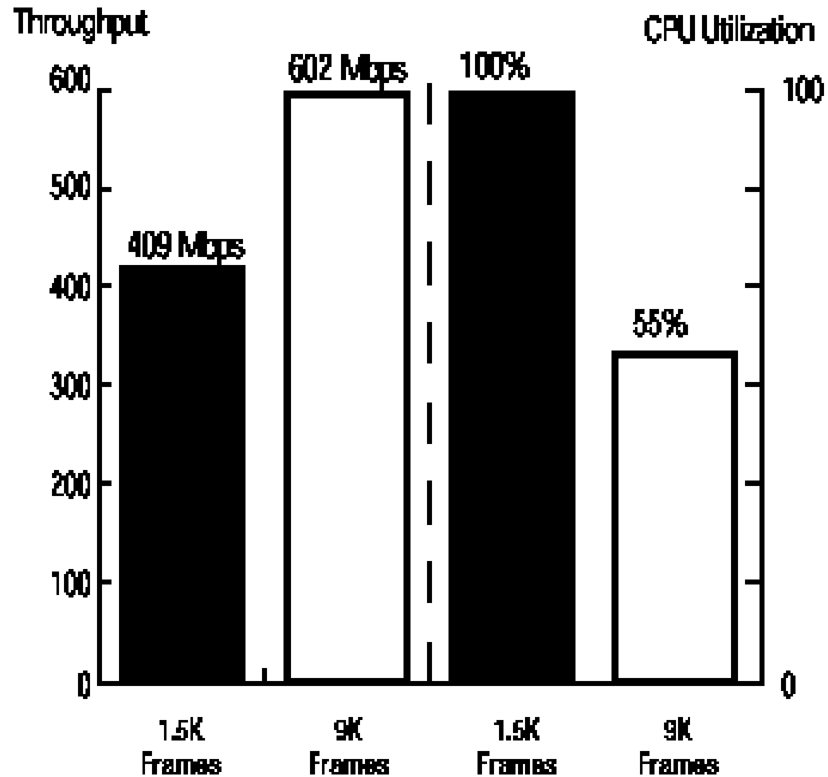


Figure 4.3: Extended (Jumbo) Ethernet Frames vs. Standard Ethernet Frames (source: Alteon Networks [Weba])

4.2.1 The Jumbo Frames Debate

We have shown that, theoretically speaking, there should be little performance benefit in increasing the payload size from the standard 1500-byte. However research conducted by Alteon Networks [Weba] indicated that by increasing the payload size from the standard 1500 bytes to 9000 bytes, a great performance benefit could be obtained, both in terms of throughput as well as CPU utilization rates (see Figure 4.3).

Alteon give a number of reasons why such a benefit is obtained. Firstly, the transmission of an 8000-byte long payload implies that the entire payload would fit exactly in two memory "pages". Rather than wasting a considerable time in moving data around, the OS would be able to simply remap the appropriate pages without

the need of copying the data word by word. In contrast, the equivalent amount of data sent using maximum length 1518 byte Ethernet frames requires six host copy operations and thus three times the host CPU cycles. As we have noted previously, as the payload gets larger, the header of each packet takes up proportionally less overhead cost. For example, parsing and building the packet header takes the same amount of time for a large packet as a small one. This means that not only is the host CPU left with more processing time, but also there is less bandwidth “wastage” on the line. Another reason given by Alteon as to why throughput would improve is the fact that some popular applications such as the network file system (NFS) used predominantly by UNIX OSes use a large datagram (8400 bytes in this case). An increase in frame size would thus mean that a single NFS datagram would fit in a single Ethernet packet.

There are, however, a number of disadvantages regarding such a change. The chief problem is that all intermediatory routers between two endpoints supporting extended frame sizes must also be configured to support larger payloads. If they are not, the router interfacing with a 1500-byte-only router may end up fragmenting all incoming packets to the standard 1.5k length, greatly increasing the load on both the router as well as the receiver, which has to re-assemble all the packets. Worse still, if the `DONT_FRAGMENT` bit is set in an IP header, the router would have no option but to drop the entire frame. In such a scenario, the router would send a message back to the sender informing the IP protocol stack to throttle back by sending packets with a lower MTU. With a great number of routers already in place, such a change would not mean a trivial upgrade scenario to the Ethernet industry. This is problematic since one of the chief aims of Gigabit Ethernet has always been to keep Ethernet as scalable as possible while changing as little as possible from the original format. Alteon Networks suggest partitioning the systems from the ones able to support the extended frame format from the ones which do not.

The graph shown in Figure 4.4 is from a study [KCT98] of traffic on the InternetMCI backbone in 1998. It shows the distribution of packet sizes flowing over a

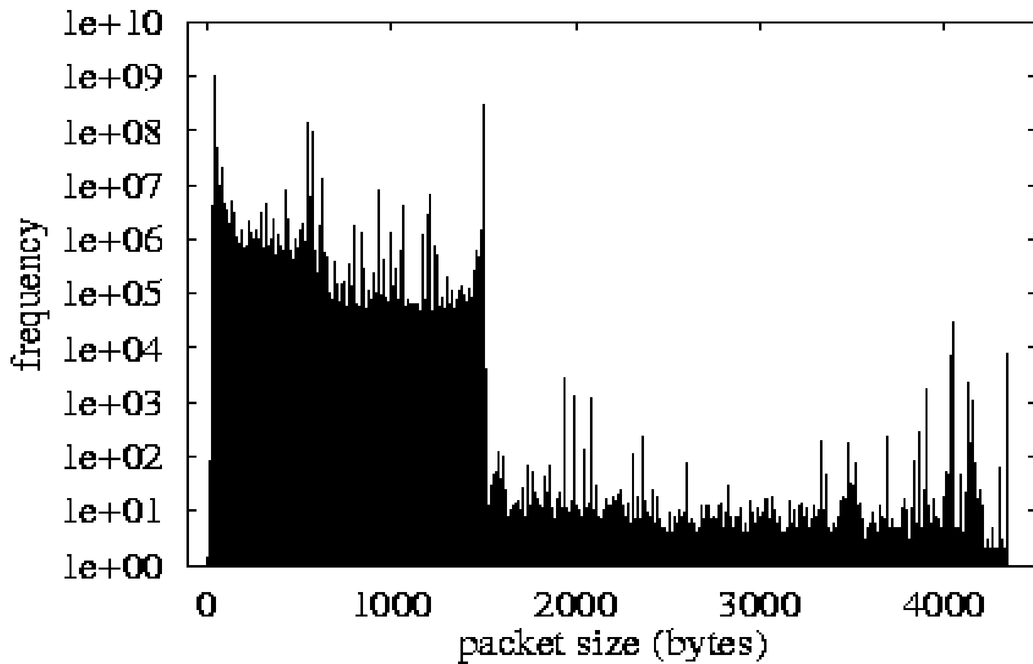


Figure 4.4: Frame sizes analysis

particular backbone OC3 link. There is clearly a wall at 1500 bytes (the Ethernet limit), but there is also traffic up to the 4000 byte FDDI MTU. However note that while the number of packets larger than 1500 bytes appears small, more than 50% of the bytes were carried by such packets because of their larger size. Also, the above traffic was limited by FDDI interfaces (thus the 4000 byte limit). Many high performance flows have been achieved over ATM WAN's offering 9180 byte MTU payloads.

Still, the IEEE is opposed to changing the established standard any time soon. We shall be keeping this debate very much in mind during the next trials by specifically comparing the two payload sizes in all our results.

4.3 Determining line speed

Having determined what should theoretically be achieved using frames of different payload lengths, our next step is to find out if the hardware is able to support the same throughput levels. The theoretical calculations do not include the costs of transferring the data from the host onto the NIC, nor do they take into consideration the time taken for all the other routines such as buffer handling. This poses a problem since we normally expect the host to construct and ultimately receive all Ethernet packets. Fortunately however, the firmware gives us the flexibility to control the hardware extensively. We therefore modify the firmware in such a way as to construct packets in the NIC itself and enqueue them onto the MAC interface as fast as possible. The firmware only fills up the appropriate header bytes (given once by the host) without bothering about the payload since its contents is entirely irrelevant to this exercise. A quick look at the technical manuals also confirmed that the MAC should be able to transmit packets out on the line up to the full 64k size (including headers). We thus set the firmware to send out packets at the fastest speed possible, using different packet sizes for each test cycle. The NIC also updates its statistics itself by incrementing the number of packets sent out on the network. Every second or so, the host software reads the elapsed time directly from the NIC's timer and calculates the actual bandwidth obtained. After taking a number of readings, the results shown in Figure 4.5 were obtained.

Payload length (bytes)	Bandwidth (Mbits/sec)
1498	975.4466
9054	995.8126

Table 4.2: Line Speed throughput for different payload sizes

Again we note that speedup is pronounced with relatively small payloads, but then tallies off at around 10k bytes (for a closer analysis, see Figure 4.6). Recall that there are no overheads related to shuttling data from the host to the NIC. Furthermore, since the Tigon employs a queue of pending transmit (and receive)

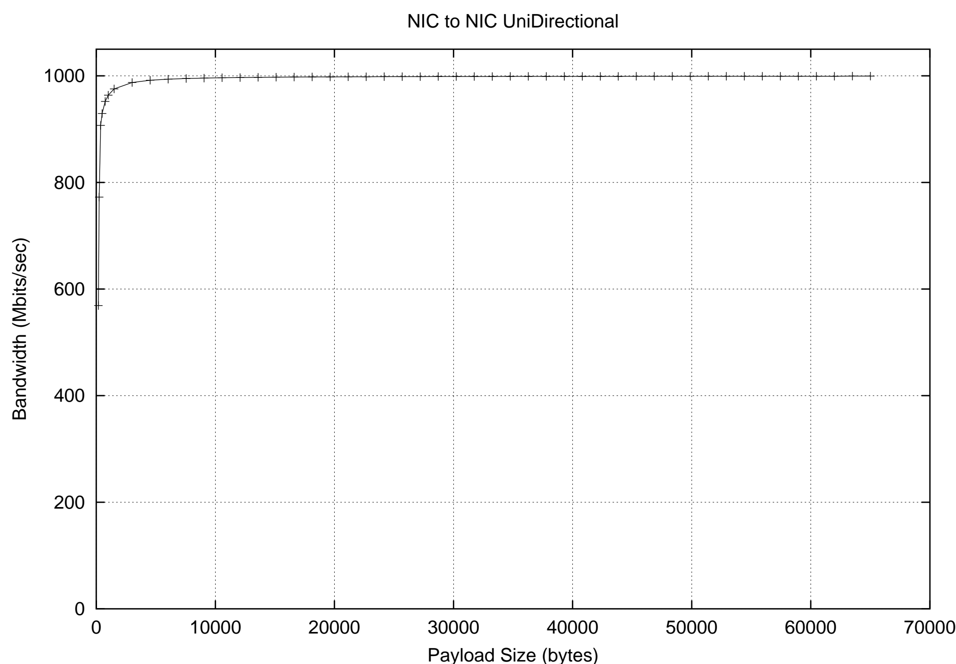


Figure 4.5: NIC to NIC UniDirectional transfer. All frames are generated by the NIC itself.

operations, the MAC should be kept continually busy without waiting for more data. When we compare it to the theoretical maximum we find a very pleasing state of affairs. Figure 4.7 clearly shows that the actual results obtained fit perfectly with our expected results. This implies that the hardware is able to cope well at Gigabit speeds, at least from the transmission point of view.

So far we have determined that :

- Large packets are not necessary for Gigabit *line* speeds to be obtained.
- The hardware can exactly mirror the results obtained by the theoretical maxima.
- The use of standard 1500-byte packets should give us a maximum bandwidth of 975Mbits/sec.
- A firmware has now been developed which sends out packets as fast as we can. This will aid us in other investigations.

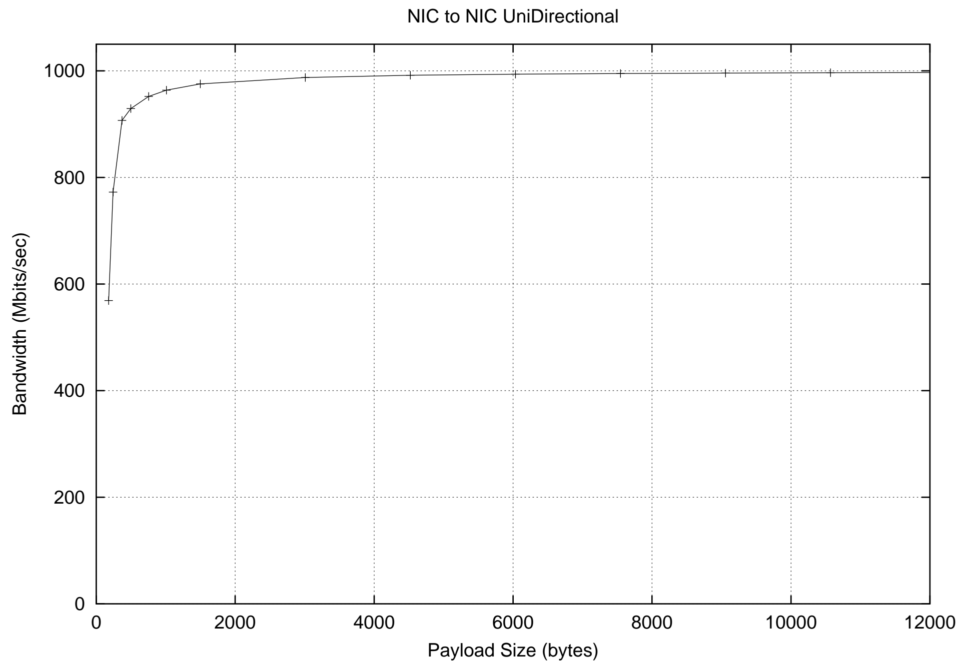


Figure 4.6: NIC to NIC UniDirectional (detail)

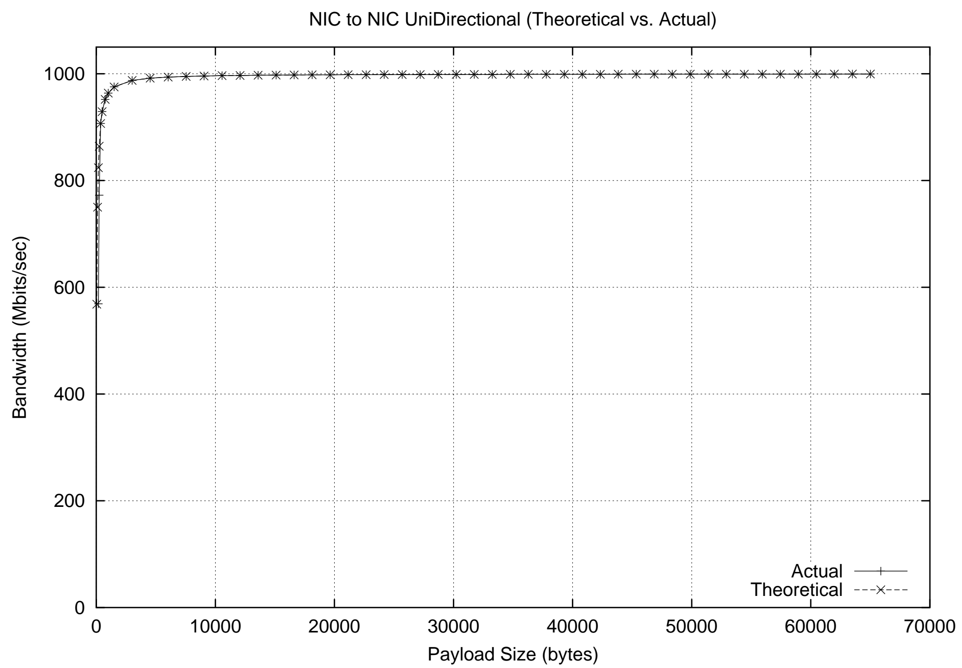


Figure 4.7: NIC to NIC (Theoretical vs. Obtained)

4.4 PCI Bus

The 975 Mbit/s seems a far cry from the 409 Mbits quoted by Alteon (see Figure 4.3). This seems odd for we were able to obtain almost the same throughput with standard frames as in larger ones. We thus turn our attention to the next potential bottleneck, the PCI bus. Recall that in §3.1 we have calculated that the maximum transfer speed utilizing a 33Mhz/32-bit PCI bus is 125.89 MB/sec. However, this theoretical speed does not take into account the cost of setting up the transfer transaction. Depending on the underlying chipset, the PCI bus may be optimized to perform better in certain transfer directions (that is, sending from the hosts' memory to the interface card may take longer than receiving data from the interface card to the host or vice-versa).

In PCI systems, there is a tradeoff between the desire to achieve low latency and the desire to achieve high bandwidth. High throughput is achieved by allowing devices to use long burst transfers. Low latency is achieved by reducing the maximum burst transfer length. Table 4.3, taken from the PCI bus specification [Gro], illustrates the effect of using different burst lengths.

Data Phases	Bytes Transferred	Total clocks	Latency Timer (clocks)	Bandwidth (MB/S)	Latency (us)
8	32	16	14	60	.48
16	64	24	22	80	.72
32	128	40	38	96	1.20
64	256	72	70	107	2.16

Table 4.3: Latency and bandwidth of different Burst length transfers.

Data Phases number of data phases completed during transaction

Bytes Transferred total number of bytes transferred during transaction (assuming 32-bit transfers)

Total Clocks total number of clocks used to complete the transfer
total_clocks = 8 + (n-1) + 1 (Idle time on bus)

Latency Timer Latency Timer value in clocks such that the Latency Timer expires in next to last data phase
latency_timer = total_clocks - 2

Bandwidth calculated bandwidth in MB/s
bandwidth = bytes_transferred / (total clocks * 30 ns)

Latency latency in microseconds introduced by transaction
latency = total clocks * 30 ns

The calculations clearly show that as the burst length increases the amount of data transferred increases. The longer the transaction (more data phases) the more efficiently the bus is being used. However, this increase in efficiency comes at the expense of larger buffers.

Since the Tigon allows us to control the DMA engines directly, we make use of this feature to benchmark the PCI bus. Any host accesses during this test may interrupt the burst transfer, so all timing is done directly by the NIC with the host only reading the elapsed time after sleeping for a number of seconds, long enough for the test to complete. To perform the tests, we extend the firmware to accept two new commands: `TG_DO_SEND_TEST` and `TG_DO_RECV_TEST`. These instruct the firmware to either commence the host to NIC DMA test or the NIC to host test.

When the host wishes to commence one of the above tests it inserts this command into the NIC's command ring together with two other parameters — the DMA transfer size and the number of packets to transfer. Since the test is only commenced when we give the explicit command, we may safely utilize our code together with the existing firmware. When the firmware receives one of these commands, it starts filling up all

available DMA descriptors[1] with the right transfer size parameter. When the NIC finishes each DMA, an event is triggered which eventually leads to a specific function being called. We modify this function so as to either insert new descriptors or else store the final timer result. This result is subtracted from an initial timer value and returned to the host via the event ring.

4.4.1 Host to NIC

In this test we are not concerned with Ethernet packet processing or transmission. All we set out to do is to test the performance of the PCI bus since this will determine if it is a factor in throughput loss.

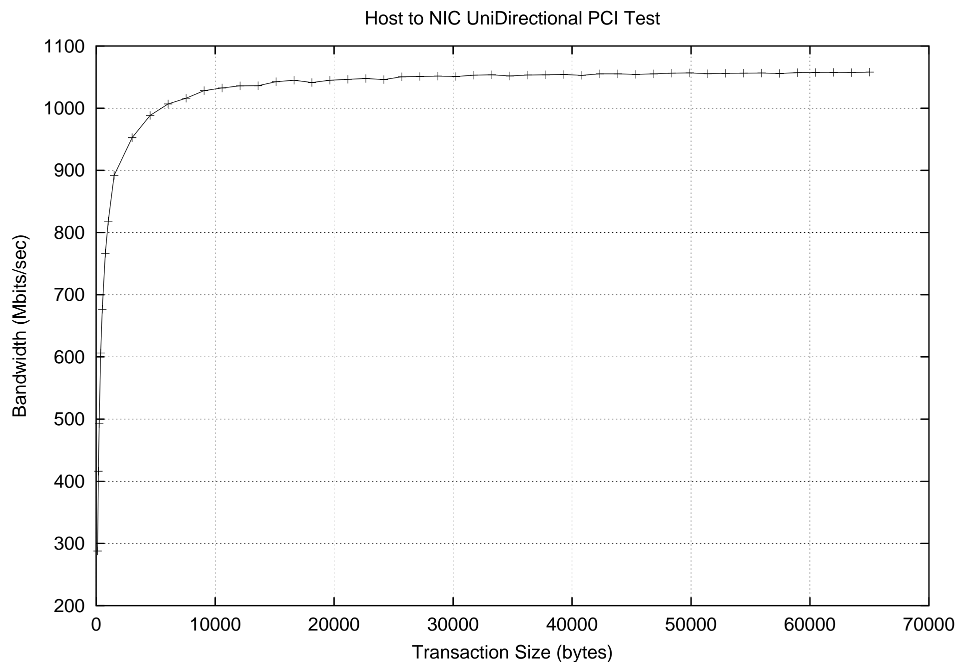


Figure 4.8: Host to NIC PCI UniDirectional

Upon a casual review of our results (see Figure 4.8 and Figure 4.9), it would seem that our near-gigabit goal can be obtained despite utilizing the slowest PCI bus va-

[1]Like send and receive descriptors, the NIC maintains a queue of pending DMA operations

riety. However a closer inspection reveals a gloomier picture.

Transfer size (bytes)	Bandwidth (Mbits/sec)
512	676.6545
768	766.6503
1024	818.2435
1512	891.9083
9072	1028.2341

Table 4.4: Host to NIC UniDirectional (fragment)

It is clear that, at smaller sized transfers, the cost of setting up the DMA transactions over the PCI bus plays plays a profound effect on the maximum bandwidth which may be obtained. At this point, the claim by Alteon that by extending the payload length a good performance benefit could be obtained, starts making sense. This test shows that, just for moving data between the host and the NIC and without even performing any kind of Ethernet packet processing, there is already a notable 137 Mbit/s difference between the standard frame sizes and the extended ones.

This test performs DMA transfers at the highest possible rate by ensuring that the DMA engines are always kept busy. We obviously expect that in normal transmission from the host this rate should drop. Still, we now have a new upper limit, reducing the possible 975 Mbit/s we obtained in the first test to an 891 Mbit/s maximum.

The next test measures the performance of doing the reverse, that is, transferring data from the NIC to the host.

4.4.2 NIC to Host

The rationale for this test is very similar to the one used by the Host to NIC test. As before, the host only makes one communication attempt with the NIC (to order the

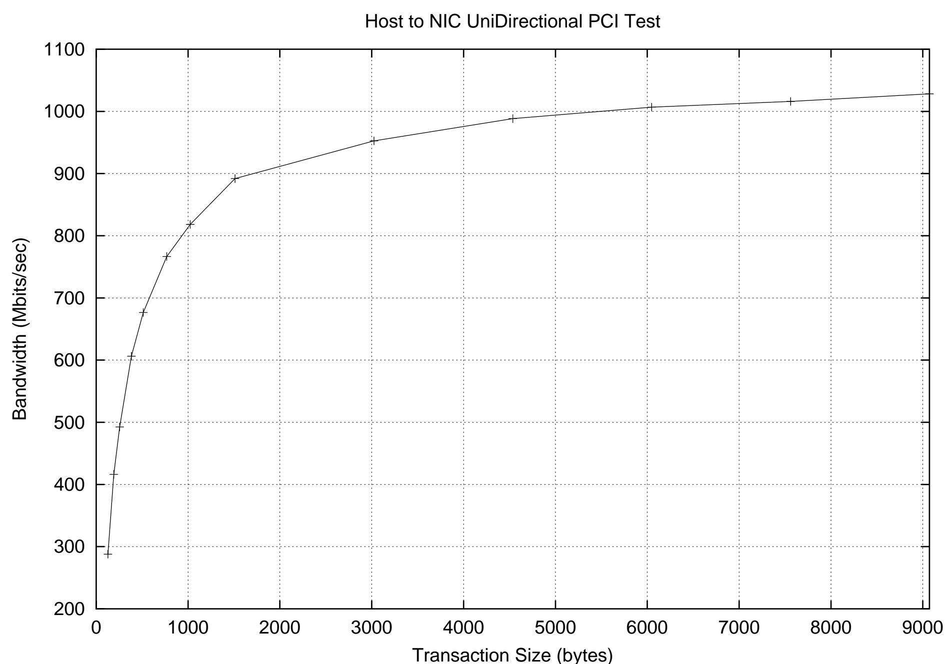


Figure 4.9: Host to NIC UniDirectional (detail)

Transfer size (byte)	Bandwidth (Mbits/sec)
1512	665.6390
9072	922.8537

Table 4.5: NIC to Host (fragment)

NIC to commence the test). This is necessary to avoid interrupting the DMA bursts.

Again we are presented with a similar picture whereby smaller transaction sizes suffer from a serious overhead penalty. One should also note that, just by using the larger 9000-byte frame size, an improvement of over 38.6% could be obtained.

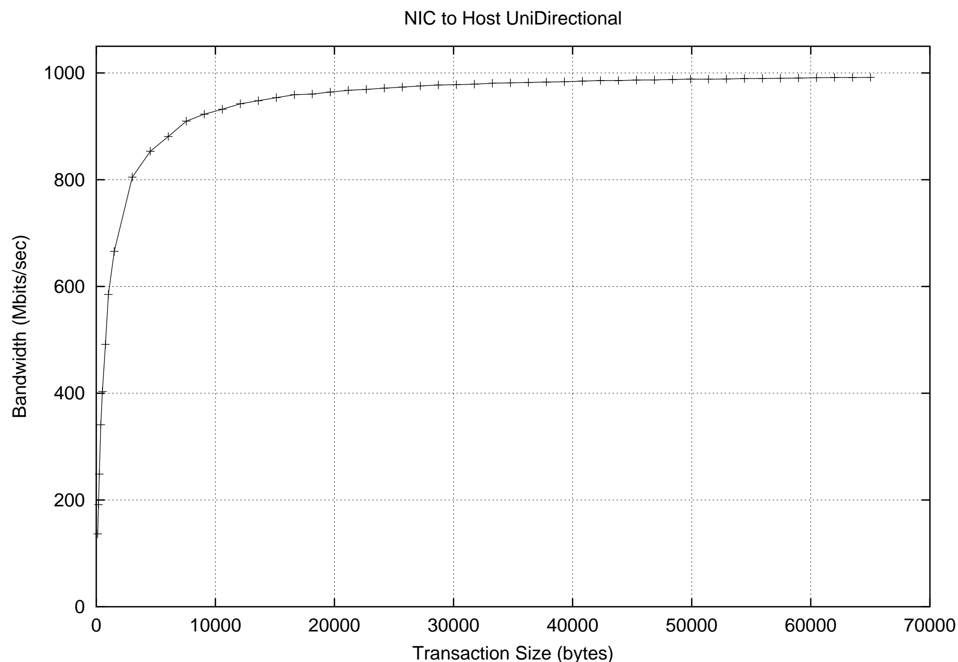


Figure 4.10: NIC to Host UniDirectional

So far we have determined that

- While large packets are not necessary for high line speeds, larger DMA transfers offer increased throughput.
- The issue of using larger frame sizes seems justified so far, in terms of performance.
- The maximum, host to host, bandwidth we can obtain on our machines using 1.5k payloads is 891 Mbits/s in the transmit direction and 665 Mbits/sec in the receive direction.
- The maximum DMA bandwidth obtained is greatly dependant on the number of bytes transferred with each transaction

---

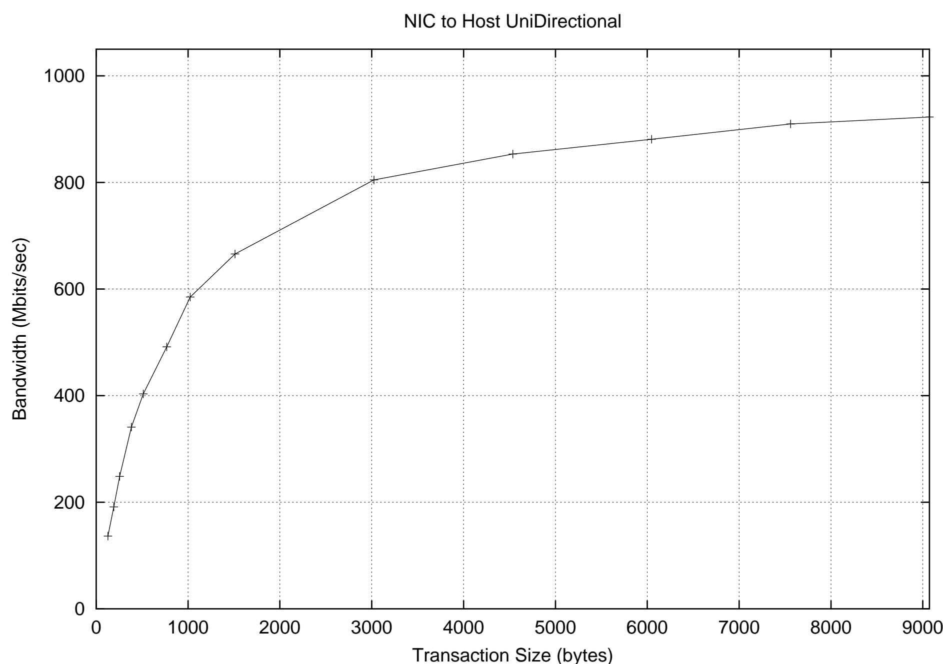


Figure 4.11: NIC to Host UniDirectional (detail)

4.5 Host transmission

The next logical step is to investigate the rate at which our host can transmit packets onto the network, in other words, we will be investigating the maximum bandwidth we can obtain by using the standard firmware and the host as the *source* of the transmitted packets. Since reception and transmission involve two differing mechanisms, we will be investigating each area in turn. Furthermore, since we are relying on different stages in the communication cycle (that is, the hosts' construction of packets, the copying of data onto the NIC, transmission, and so on) we shall be dividing this test into two stages. Firstly, we shall set up the NIC to either generate packets with the receiver transferring all received packets onto the host; or alternately we shall set the NIC to receive the packets originating from a host.

We can do this kind of test with the certainty that any bottlenecks, if present, are originating from the host to NIC interaction. We can be sure of this because we have already determined (in §4.3), that the physical transmission step runs at the maximum throughput possible. Once we determine the maximum transfer rate of

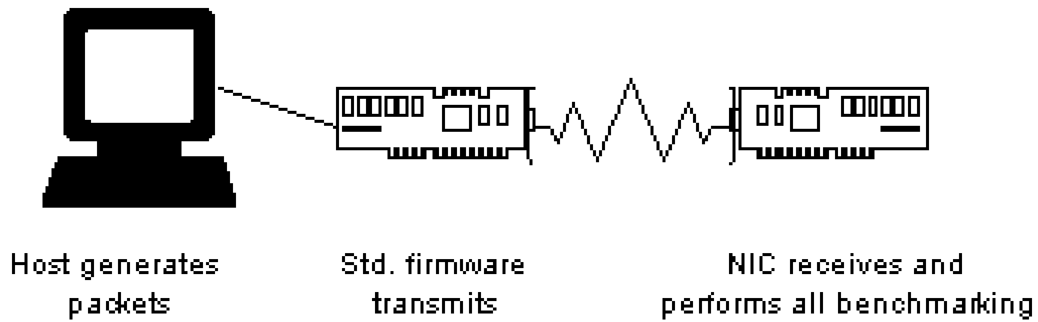


Figure 4.12: Testing standard firmware (TX)

both sender and receiver we would then be able to proceed to next step; that of host to host transmission.

4.5.1 Testing transmission from a host source

In this test, we shall be testing the maximum throughput we can obtain when using the standard firmware and using the host as the source for all our packets and the NIC as our receiver (see Figure 4.12). We already have the firmware for fast reception of frames for it was used in §4.3. As for the sender, the host is the only provider of packets. However, since we want to exclusively test the maximum throughput possible when utilizing the standard firmware, we only ever create an Ethernet packet once and transmit it repetitively for a number of seconds. Even before commencing the test we already know the maximum bandwidth we can ever hope to obtain, since this was found out in the PCI (DMA) test in §4.4. Without further ado, we hereby present our results.

Transfer size (bytes)	Bandwidth (Mbits/sec)
1498	808.9748
9058	995.8060

Table 4.6: Host transmitter, card receiver

It is evident that a clear pattern is emerging here. In all tests we have performed

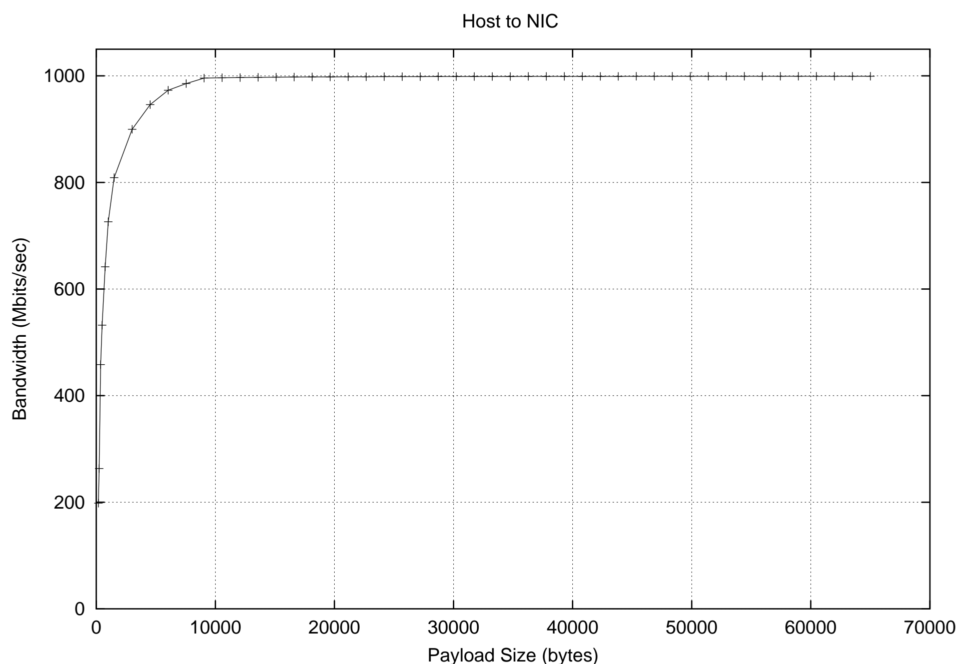


Figure 4.13: Host transmitting with the NIC benchmarking at the other end.

so far, we have seen a good speedup as we increase the total frame size, only to level off beyond a certain threshold (around 10k). When we compare the results of using the standard 1.5k packets with the extended frames we can again reconfirm that the extended frame size gives a performance benefit of over 186 Mbits/s. This is shown in more detail in Figure 4.14

A more interesting analysis can be found in figures 4.15 and 4.16 whereby we compare the performance of the bandwidth obtained with the throughput of the our PCI bus (which we found out earlier). Note that the difference between the two graphs in Figure 4.15 reflects the extra work performed by the NIC in order to process the packets to send them out on the network. Also note that at 1000 Mbits/s, the host to NIC graph suddenly levels off since the transmission limit has been reached. This contrasts with the PCI throughput obtained whereby a further performance boost is registered.

In Figure 4.16 we highlight the difference in bandwidth between the performance of the PCI bus and the bandwidth received and plot each point. Note that, with small

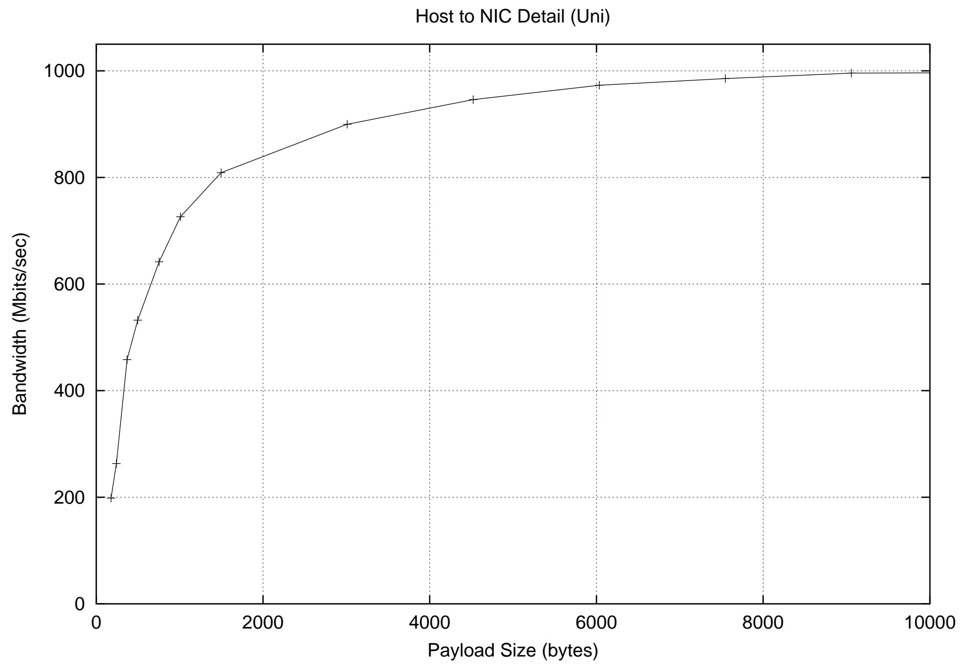


Figure 4.14: Host transmitting, NIC receiver benchmarking (detail)

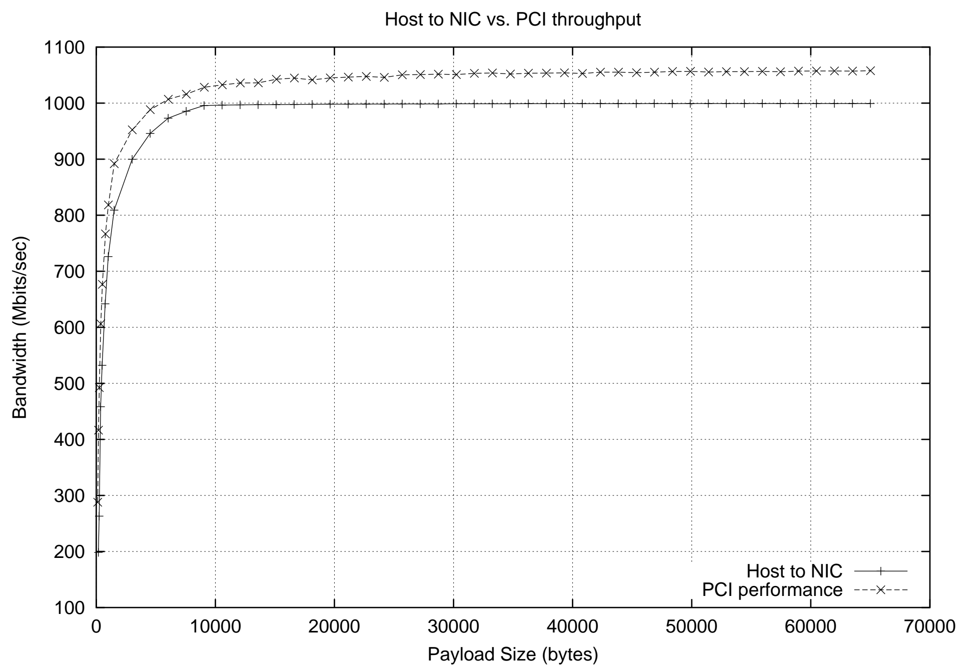


Figure 4.15: Host transmitting, NIC receiver benchmarking vs. PCI performance

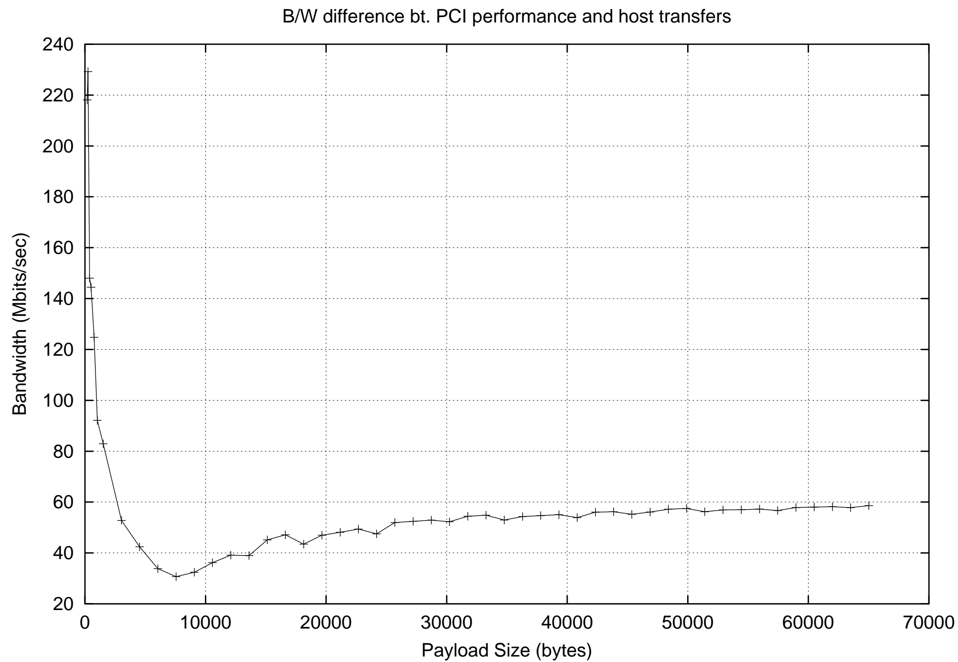


Figure 4.16: Bandwidth difference between Host transmit and DMA performance

packets, the NIC is tied up processing many packets with the end result being that the host is forced to enqueue packets at a slower rate. As the packet length increases, a balance is reached with the NIC coping well with all the packets received leading to a drop in performance of just 30Mbit/s. Unsurprisingly, this "ideal" packet size lies in the 7-10k range. As the packet size is further increased, the PCI performance continues to increase, but, as can be evidenced in Figure 4.16, the performance speedup levels off since the Ethernet transmission limit has been reached.

> In transmission, when compared to the PCI throughput and when the NIC is using standard firmware, approximately 32Mbits/sec is lost when processing extended frames and around 83Mbits/s when processing standard 1.5k frames.

4.5.2 Testing host reception from a NIC source

Insofar we have determined that our maximum transmission rate using standard 1.5k packets was set at 975 Mbits/sec. This was subsequently revised downwards to 891 Mbits/sec when the PCI throughput was taken into account and revised downwards again to 808 Mbits/sec for normal firmware to process packets and send them out on the network.

The next step is to check the other end of the message cycle. We want to determine how fast the NIC's standard firmware is able to process packets and send them to the host. We want to make sure that we receive packets as fast as possible for, if the sender's rate drops below what we can achieve, our results will be distorted. This problem has already been solved efficiently before and we therefore adopt the same solution by making use of the firmware we utilised in §4.3 when testing the NIC to NIC line speed. Figure 4.17 makes this clearer.

Figure 4.17: Testing standard firmware (RX)

An analysis of results reveals a picture somewhat similar to the one presented above, only more dramatic, since more work is done on the receiver side than on the sender side. The detail on Figure 4.19 highlights the big throughput difference between the standard 1.5k packets, where performance drops to a dismal 537 Mbits/sec, and the extended frames which manage 893 Mbits/sec.

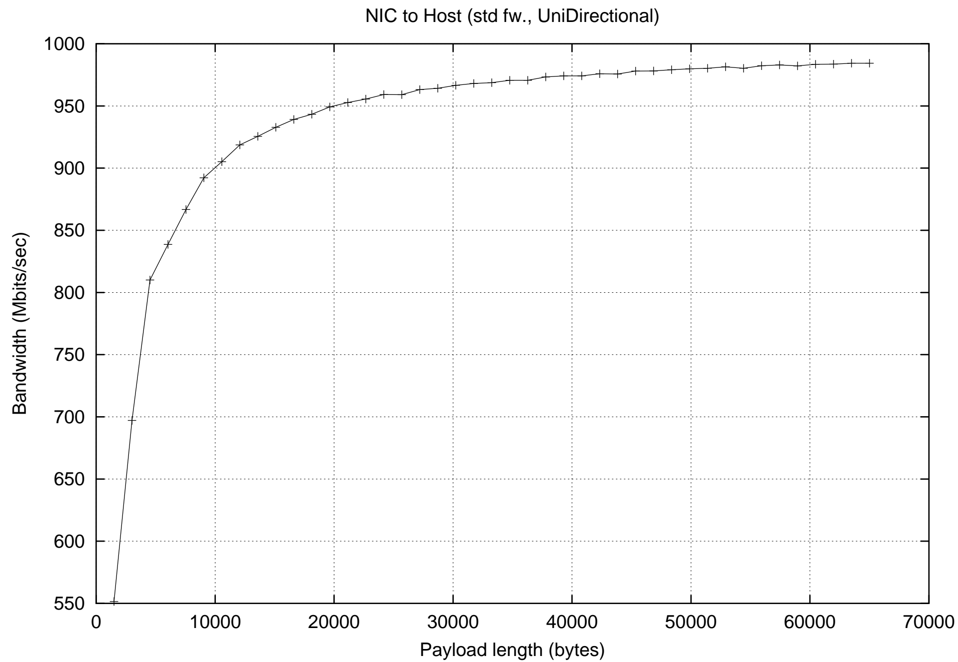


Figure 4.18: NIC to Host using standard firmware

Figure 4.20 highlights an important fact we have already noticed before — performance is greatly dependant on the underlying PCI bus speed. Notice how the two graphs almost mirror each other, the difference between them representing the extra work required to process the packets. Also note that at smaller packets, the drop in performance is substantial as the NIC has to process a large number of packets.

4.5.3 Measuring Host to Host performance

At this point we are in a position to test out the complete message cycle, that is, transmission from one host to the next. Obviously since we are benchmarking the time taken for a packet to arrive at it's destination, all timings will be performed by the receiving host.

Here we should point out that the other host under test is able to cope well with the throughputs generated. We give one final result, that of comparing the bandwidth obtained to the PCI throughput.

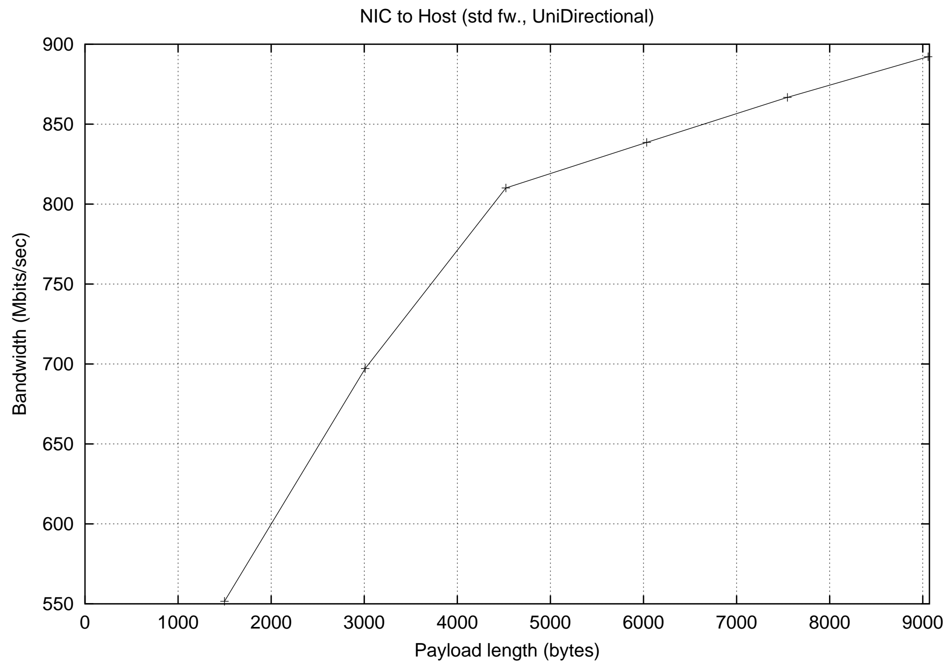


Figure 4.19: NIC to Host using standard firmware (detail)

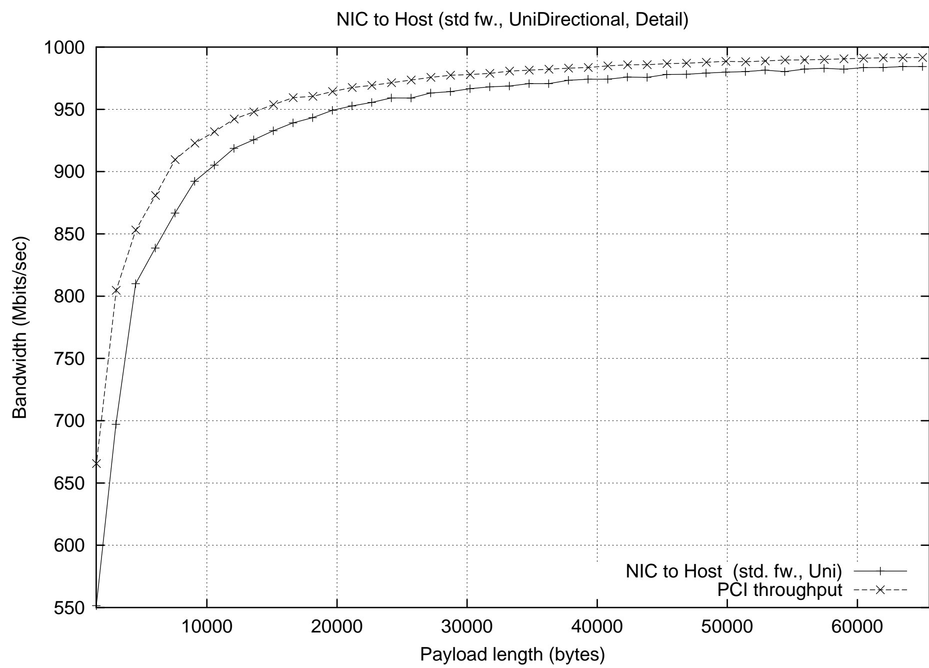


Figure 4.20: NIC to Host using standard firmware (detail) vs. PCI Performance

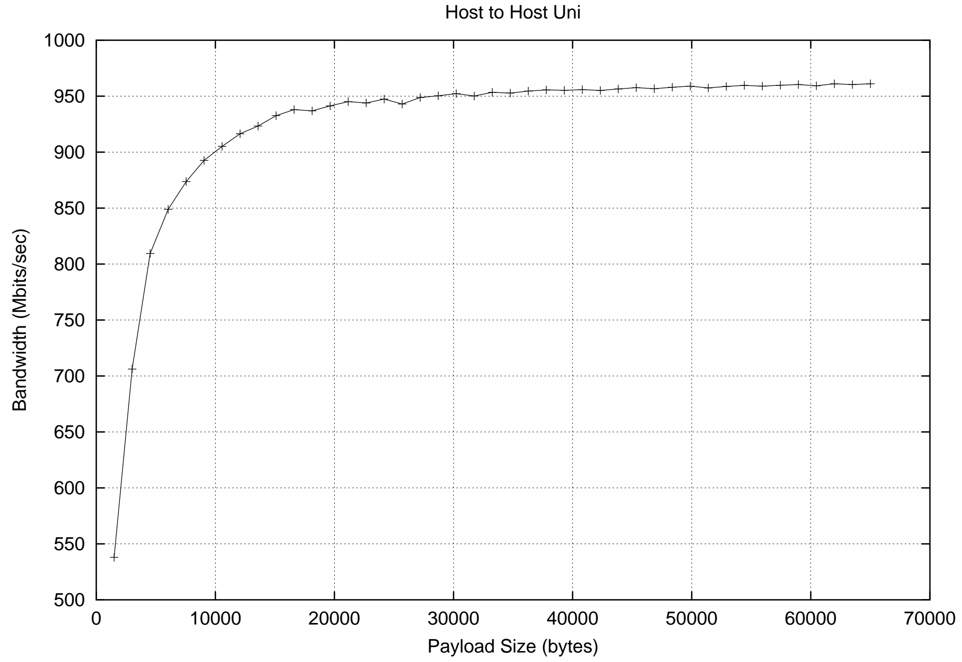


Figure 4.21: Host to Host Unidirectional transfer

Payload size (bytes)	Bandwidth (Mbits/sec)
1498	537.9818
9058	892.5703

Table 4.7: Host to host (fragment)

Once again note that the throughput we obtain is highly dependant on our PCI performance. We are now in a position to state that one of the major reasons why throughput drops so significantly when utilizing small packets is due to the performance of the underlying PCI bus.

Figures 4.23 and 4.24 summarize our results so far.

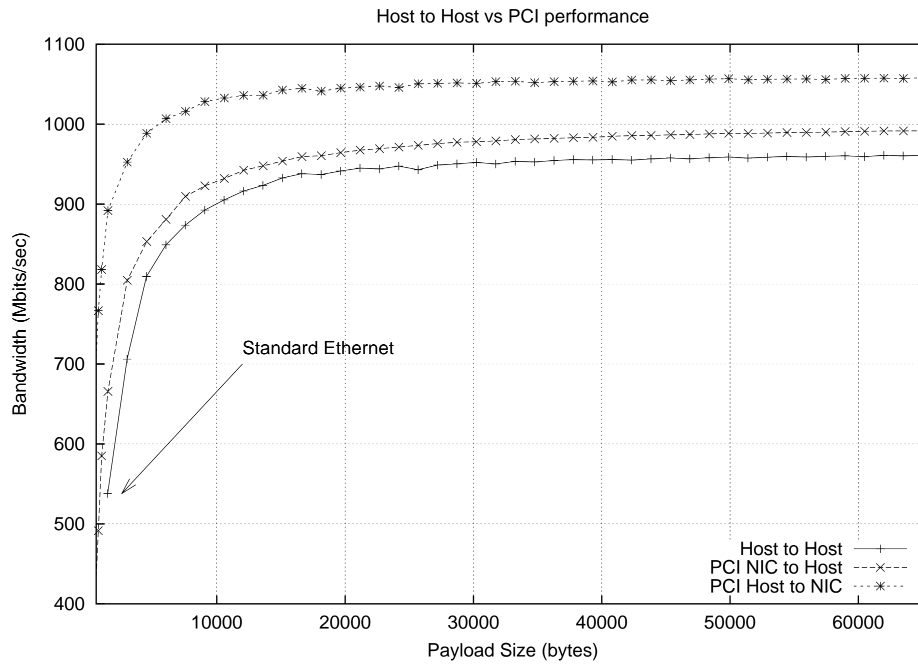


Figure 4.22: Host to Host Unidirectional vs PCI performance

Conclusions so far:

- Line speed is affected by packet size, but serious performance loss only show up when the payload lengths are small.
- DMA startup costs are expensive and affect PCI performance
- Extended (Jumbo) packets offer improved performance since PCI throughput is better when larger transactions are involved.

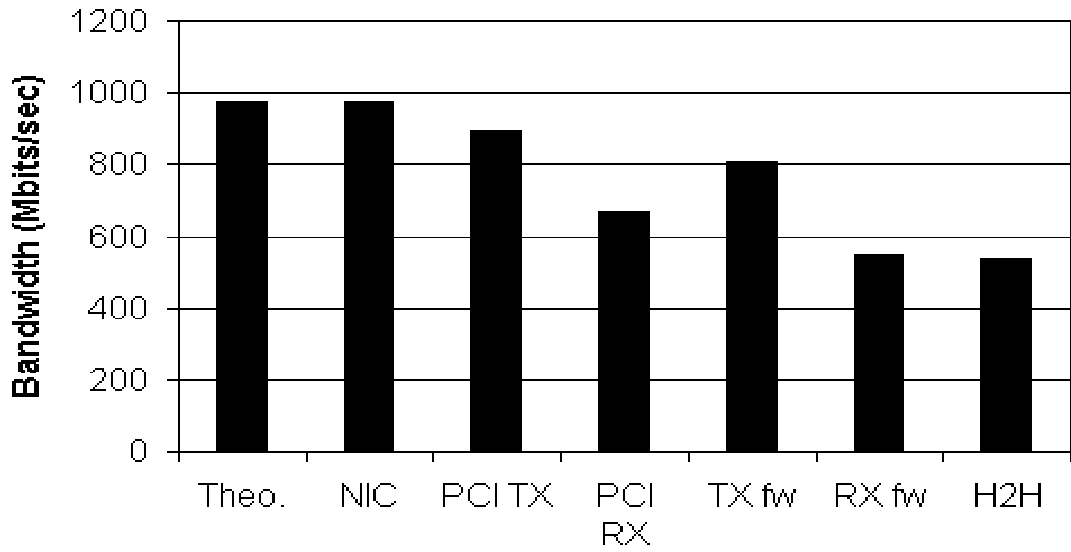


Figure 4.23: Summary of results (1.5K payloads)

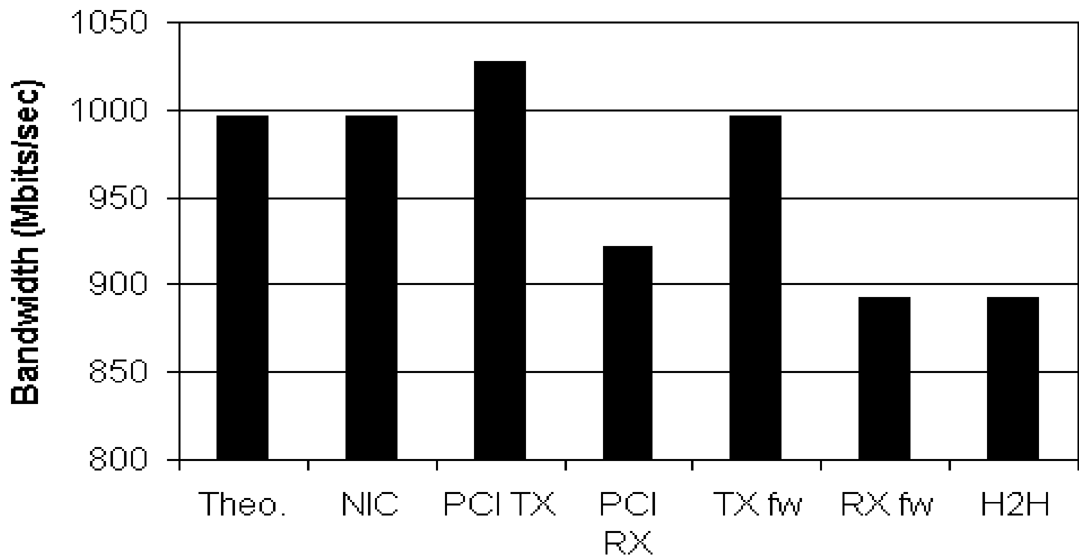


Figure 4.24: Summary of results (9k payloads)

Chapter 5

Packet Fragmentation and Coalescing

In the previous chapter we have identified the PCI as being the chief bottleneck in the communication cycle. We have also acknowledged that the standard packet length on the physical transmission part still gains a respectable throughput. In this chapter we demonstrate two techniques which, when applied, maintain the compatibility of existing systems by using 1.5k payloads while at the same time taking advantage of the higher throughput offered by larger PCI transfers.

The usual practice in all NICs is to perform a DMA request for every packet received or sent. For example, when the NIC receives a 1.5k packet it first stores it in its onboard buffers and then issues a DMA request *of the same size* to transfer the packet received to the host. It is at this stage that throughput drops dramatically because the MAC is forced to wait for the DMA request to complete.

We propose two techniques for the transmitting and receiving side which we call *packet fragmentation* and *packet coalescing* respectively; both of which we describe below.

5.1 Packet Fragmentation

Recall that in order for a packet to be successfully transmitted and received at the other end, all that is required is a valid Ethernet header followed by a number of bytes representing a payload. Normally, a host issues a single DMA request for a single frame. With packet fragmentation, the host issues a larger DMA request containing a number of packets inside it. The firmware then "breaks up" the packets and enqueues them onto the MAC queue for transmission. Figure 5.1 makes this setup clearer.

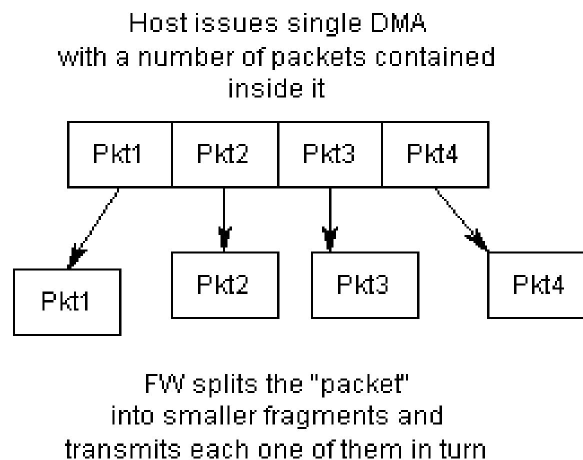


Figure 5.1: Packet Fragmentation Block diagram

Each send descriptor contains a type field which is used to signal some conditions to the NIC. By defining a new type, we make it possible for our customized firmware to work alongside the existing standard one. In other words, applications or drivers aware of the feature can make use of it while the rest can still use the standard routines as usual.

5.1.1 Results

Apart from the single host to NIC DMA transfer, the NIC does not perform any other memory copies whatsoever. This is crucial, since memory to memory copies are sure to incur performance penalties. Unfortunately, the Tigon's MAC requires

that each Ethernet packet starts on a doubleword boundary[1]. This means that, in order to avoid memory to memory copies and without further hardware support, we enforce this restriction onto the host; that is, we assume that the host always uses payload lengths exactly divisible by 8. This is the reason why we have used frame lengths of 1512 bytes (that is, a 1498 byte payload) in all our earlier tests. What we have thus achieved is the ability to *optionally* transfer up to 43 packets in one go (since the maximum DMA length is set at 64k) to take advantage of the higher PCI throughput while using the standard 1.5k frames across the network.

Algorithm 1 Fragmentation Algorithm (minor details omitted)

if send descriptor has our type bit set **then**
  $n \leftarrow (packet_length/fragment_size)$
  $p \leftarrow (packet_location)$
  for $i = 0$ to $n - 1$ **do**
    $address \leftarrow$ (p + (i*fragment_size))
    Enqueue onto MAC a descriptor of length *fragment_size* starting at *address*
  end for
end if

In theory, a performance benefit should be obtained, but how well does it perform in practice? Take a look at the figures below. In Figure 5.2 we can see a very marked improvement with a rapid speedup throughout. After just 5 fragments, Ethernet's theoretical limit has been reached (remember that we are sending out 1.5k packets on the line) so the bandwidth obtained suddenly levels off.

When we calculate the actual DMA transferred and plot the results together with what we obtained when testing out Host to NIC transfers, we find another near-perfect match (see Figure 5.4).

Once again we are in a position to shuttle packets out onto the network at the maximum possible rate. However, the receiver is still subject to throughput loss. We will be investigating a different kind of technique to tackle this problem in Section 5.2.

[1] 8 bytes since the Tigon internally uses a 64-bit memory bus

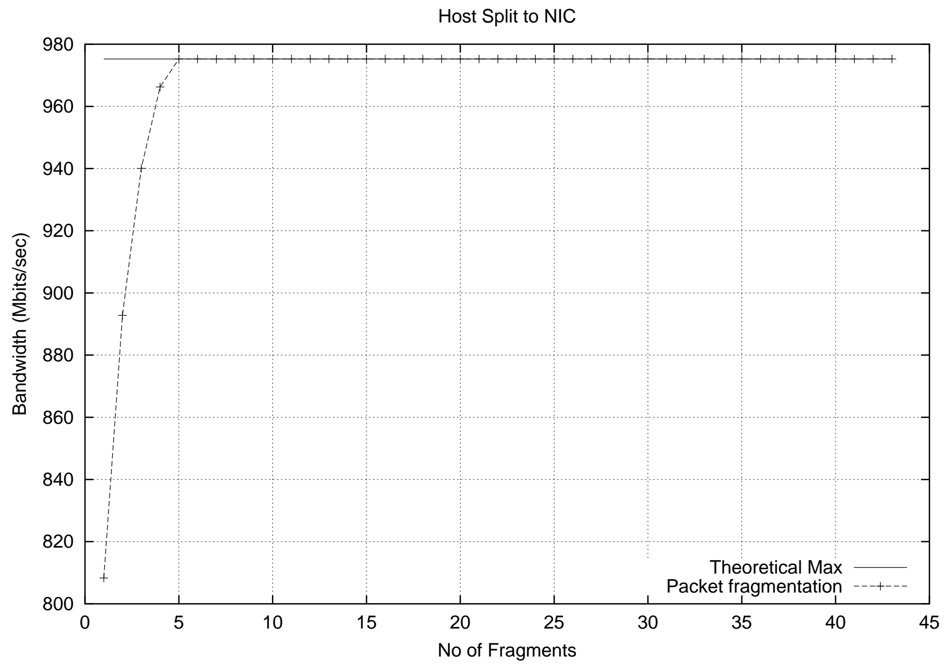


Figure 5.2: Packet Fragmentation results

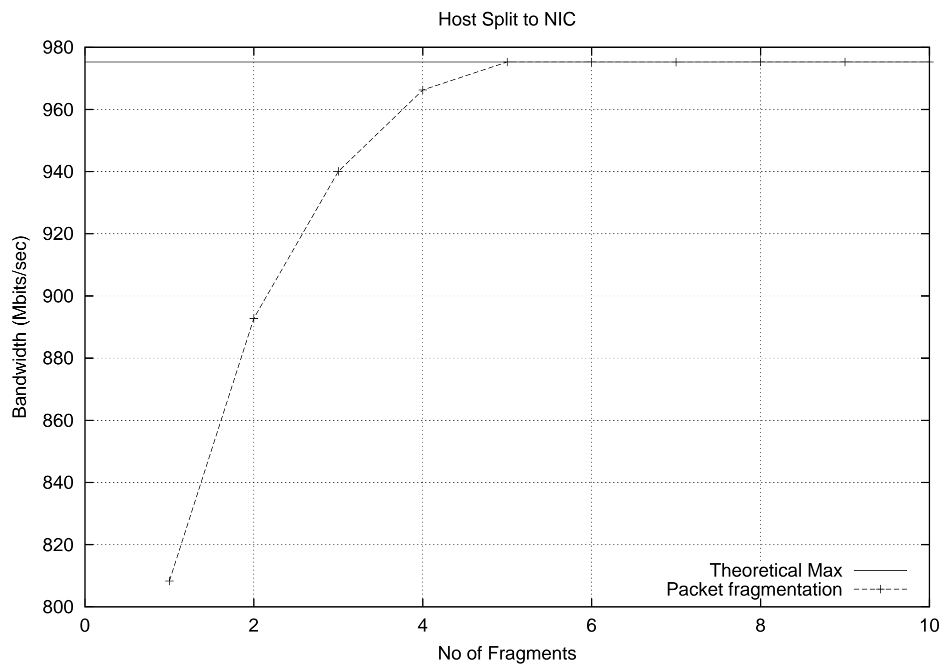


Figure 5.3: Packet Fragmentation results (detail)

Number of Fragments	Payload (bytes)	Bandwidth (Mbits/sec)
1	1498	808.3237
2	2996	892.7797
3	4494	940.0449
4	5992	966.2615
5	7490	975.2454

Table 5.1: Fragmentation results

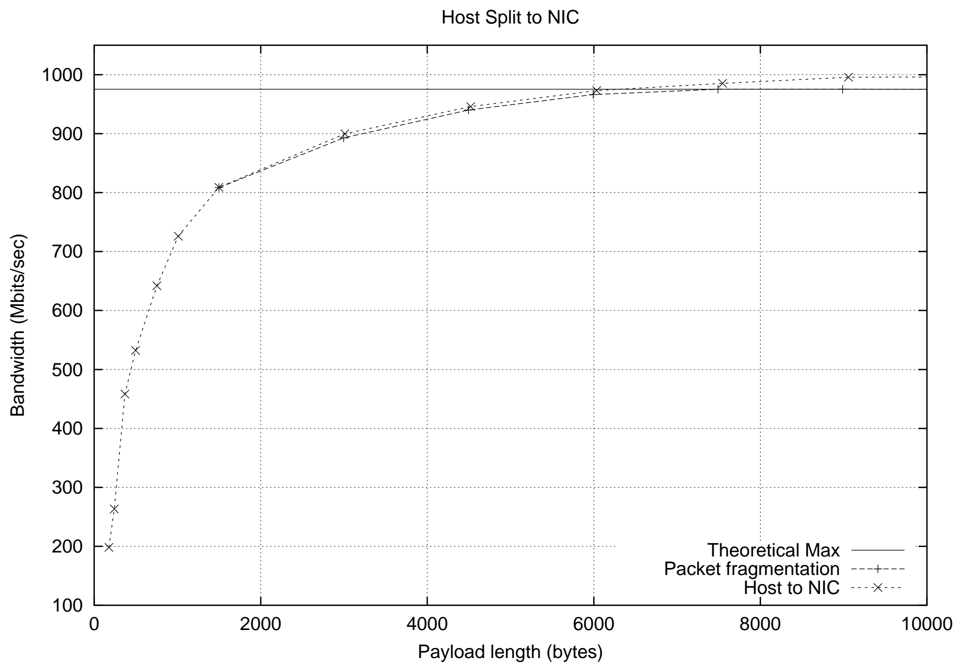


Figure 5.4: Packet Fragmentation vs Host to NIC

5.1.2 The Pitfalls

Despite the big improvement in performance, there are some drawbacks. Firstly, the host is bound to perform extra work to construct each packet and make sure that each fragment starts on a double-word boundary. Secondly, this model assumes a single-process scenario. Ideally, if there are a number of processes each trying to transmit

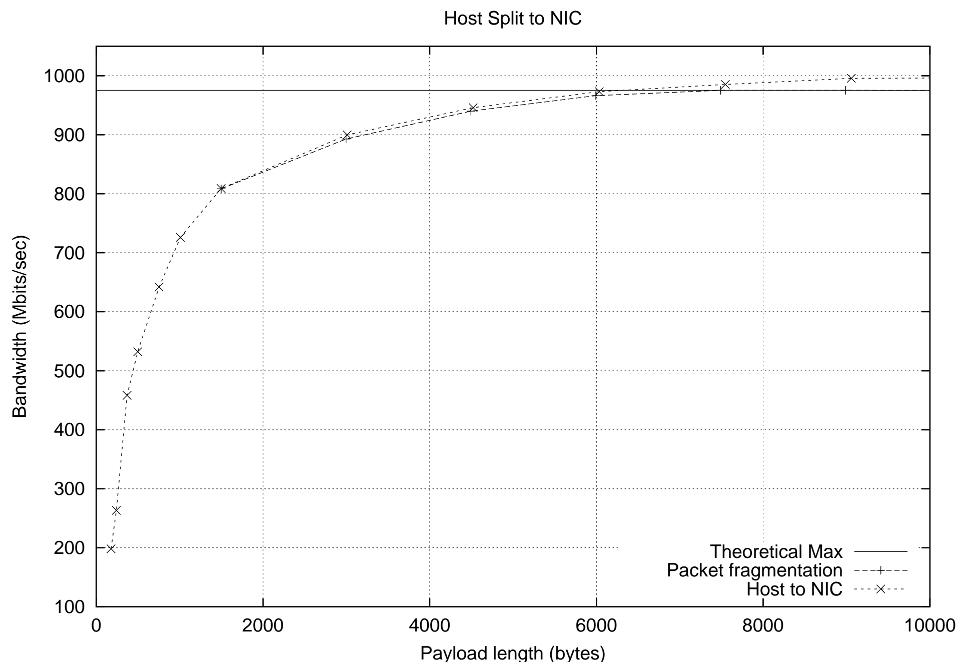


Figure 5.5: Packet Fragmentation vs Host to NIC (detail)

a single 1.5k packet, it would be more efficient to group the packets together and perform the transfer once. This implies the need of synchronization at some point, with all the ramifications that brings about. However this is an issue which affects extended frame users too.

5.1.3 Future Work

The requirement to align packets on doubleword boundaries is hardware enforced and there is little which can be done to avoid it.

What can be improved is the offloading of copying the headers when the destination address is the same. Currently, if a host wishes to send, say, 42 packets to the same destination, it has to copy the header repetitively. Ideally, the host would be able to transfer a large block of data with the firmware inserting the headers at the appropriate places. The Systems Software Research Group (SSRG) is currently researching ways of avoiding this step efficiently[2].

[2] For more information regarding SSRG, a locally set up group, visit www.cs.um.edu.mt/∼ssrg

Finally we may also extend the firmware to fragment frames based on the length of the underlying Ethernet packet rather than use the fixed 1498 byte value as we currently do.

> Packet fragmentation enables a host to achieve the maximum possible transmission throughput while utilizing standard Ethernet packets

5.2 Packet Coalescing

With the new fragmenting technique, we are now presented with a scenario whereby the only bottleneck in achieving near-Gigabit speeds lies on the receiver side, or more specifically, the transfer of packets from the NIC to the host.

Recall that in the packet fragmentation technique we have made use of a large DMA transfer to transport a number of packets. In this packet coalescing technique we delay transfers from the NIC to the host until a host-specified number of packets have been coalesced. Normally, when the host constructs the usual receive descriptors, it allocates just enough space for a standard or extended packet to be received. In this technique, the host allocates enough space for *a number* of packets. It also sets a special flag in the descriptor which is used to signal to the firmware that the next few packets are to be coalesced. As before, this system allows this technique to coexist with the standard routines.

When packets start being received, the NIC starts looking at available descriptors. If the descriptor contains our flag, the firmware updates all the usual pointers but does not immediately start DMA-ing the received frame to the host. For each received packet, the remaining length of the original buffer descriptor is updated to reflect the new packet. When there is no more space to accommodate any further packets, the NIC issues the single DMA transfer which may contain a maximum of 42 packets (if all the packets are 1.5k bytes long). The host then picks up each packet just as if it

had received them one by one.

Since we cannot predict how many packets are to be received, the NIC may end up waiting indefinitely for more packets prior to moving the packets already waiting to be transferred. For example if the host specifies that 42 fragments are to be coalesced and the NIC receives just 4 fragments, the host may never receive the 4 packets received since the NIC would still be waiting for the next packets to arrive. To circumvent this problem and also to let the host control latency, when a packet is received, the onboard timer is set to trigger a short while later. If this alarm expires and the NIC is still waiting, the firmware transfers the currently coalesced packets and the whole process starts again. Since each received packet resets the timer, the host can control the maximum time it is prepared to wait for more packets to arrive after the last one has been received.

5.2.1 Results

Take a look at Figure 5.6. Even if we do not coalesce many packets, we are still able to reach the Ethernet theoretical limit.

In Figure 5.8 we compare our results with the PCI performance, in other words, we show the extra work done by the firmware. Notice that, with small packets, the firmware keeps waiting for the DMA operations to be complete. As we coalesce more and more packets, the NIC performs large DMA transfers so the performance loss drops down to around 40 Mbits/sec. Beyond the 20k mark one can see a gradual rise again. This is because the Ethernet limit has been reached but the PCI is still able to transfer data at a faster rate. Also note that since the usual host machine under test is not able to achieve high PCI rates (for the NIC to host side), the graphs shown in figures 5.7 and 5.8 plot PCI benchmarks of the second host machine used in our tests.

The nicest thing about this system is that it does not enforce any constraints on the sender. In other words, so long as the sender complies with the Ethernet standard, the receiver is able to register a performance boost. In Figure 5.9 we demonstrate

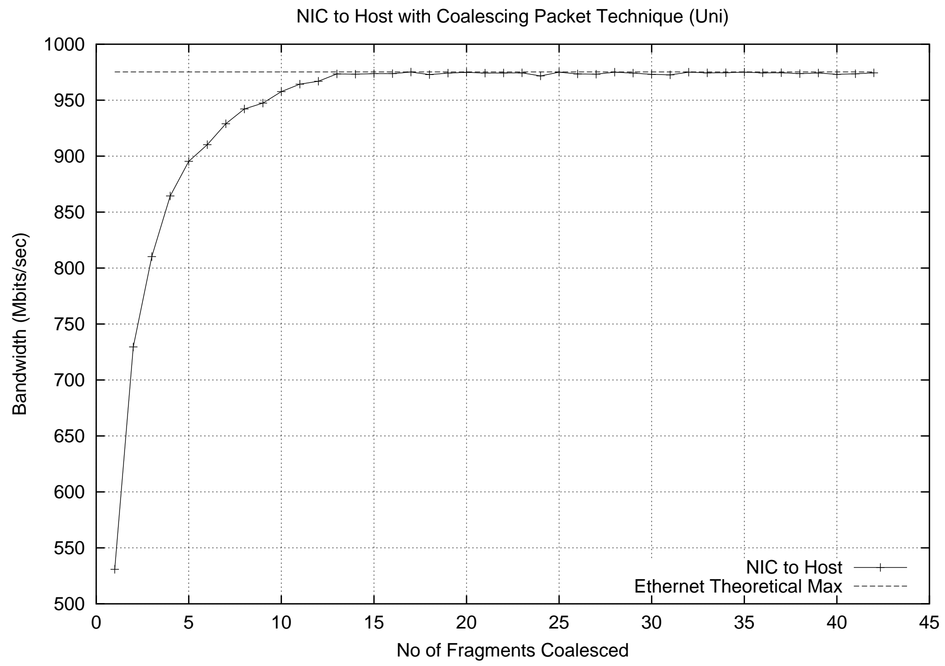


Figure 5.6: Packet Coalescing

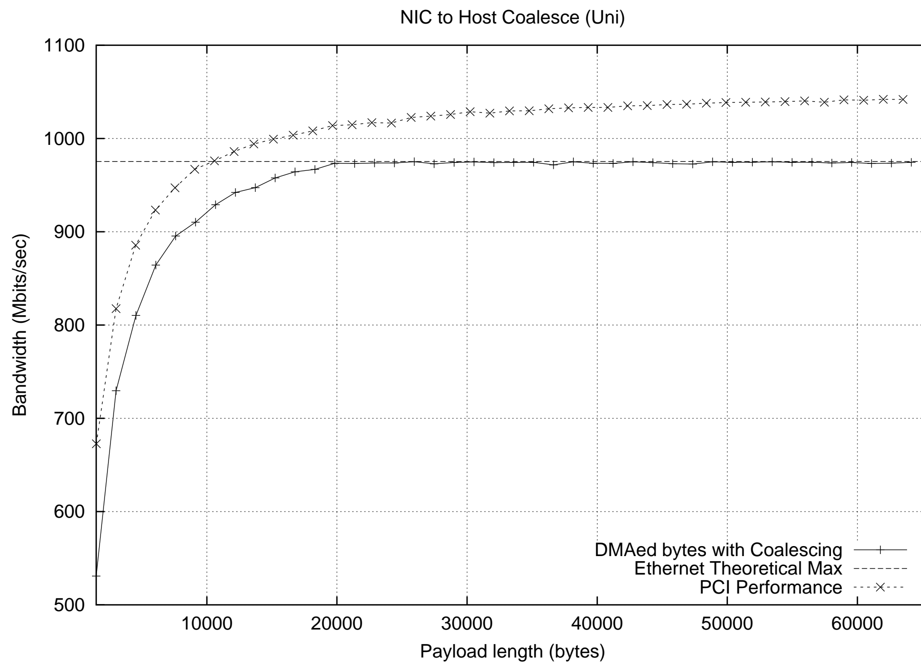


Figure 5.7: Packet Coalescing vs PCI performance

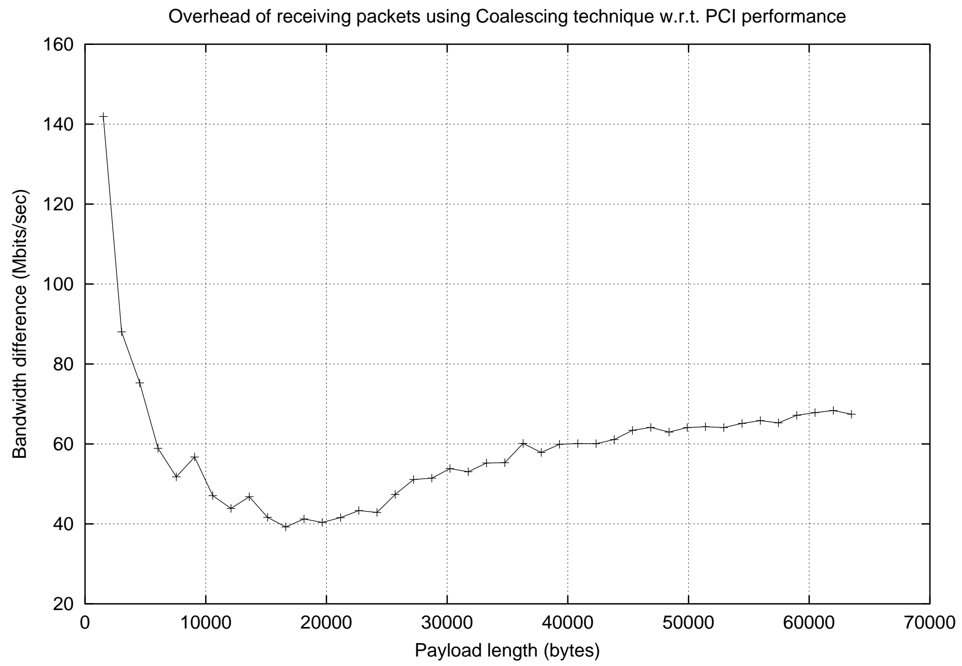


Figure 5.8: Bandwidth loss when compared to PCI performance

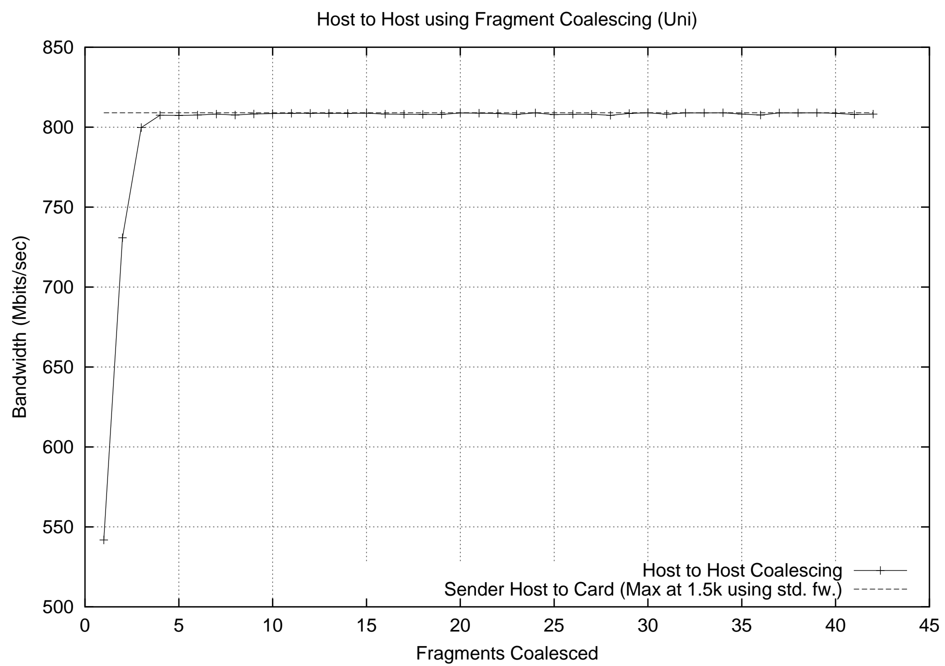


Figure 5.9: Host to Host with Packet Coalescing

how the receiver, gains such a performance benefit despite the presence of a slow transmitting host. In this setup, the host at the sender side is set to transmit a stream of 1.5k packets using the standard firmware while we switch on packet coalescing on the receiver side.

5.3 Host Fragmentation to Host Coalescing

It is now appropriate to try out both techniques described in this chapter to perform host to host transfers. In this test, we vary the number of split fragments at the sender side and the number of packets coalesced at the other end each time measuring the performance obtained. In this scenario, both hosts are able to communicate with the NIC using any DMA length while leaving it up to the NIC to communicate between one endpoint and the next using the standard 1.5k packets (see Figure 5.10).

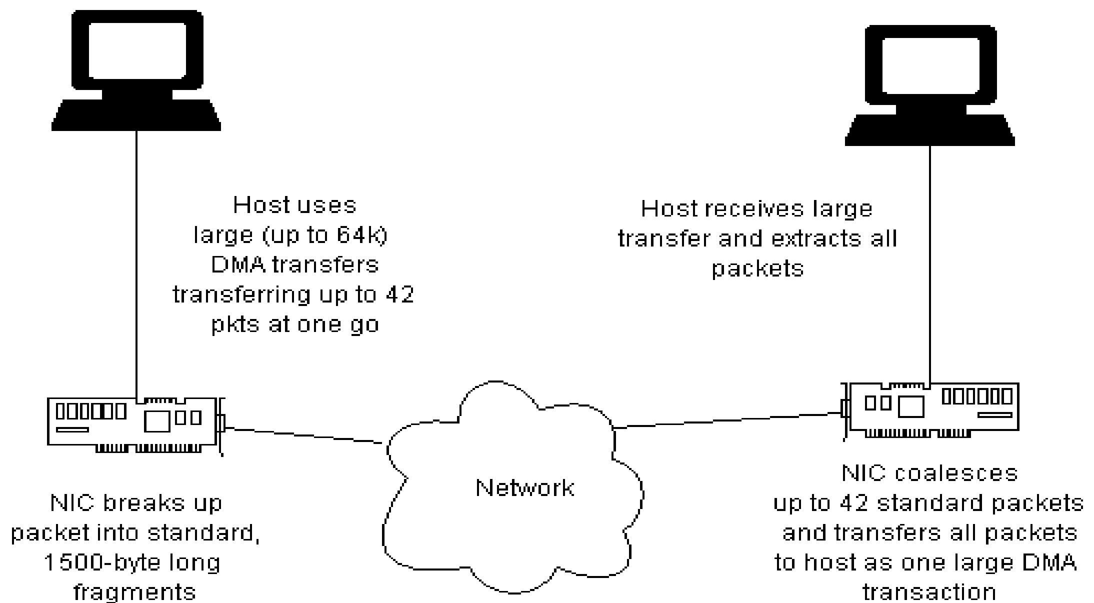


Figure 5.10: Host Split to Host Coalescing (block diagram)

An analysis of the results obtained confirm that, without compromising the 1.5k-per-packet standard, the theoretical limit has still been reached (see Figure 5.11).

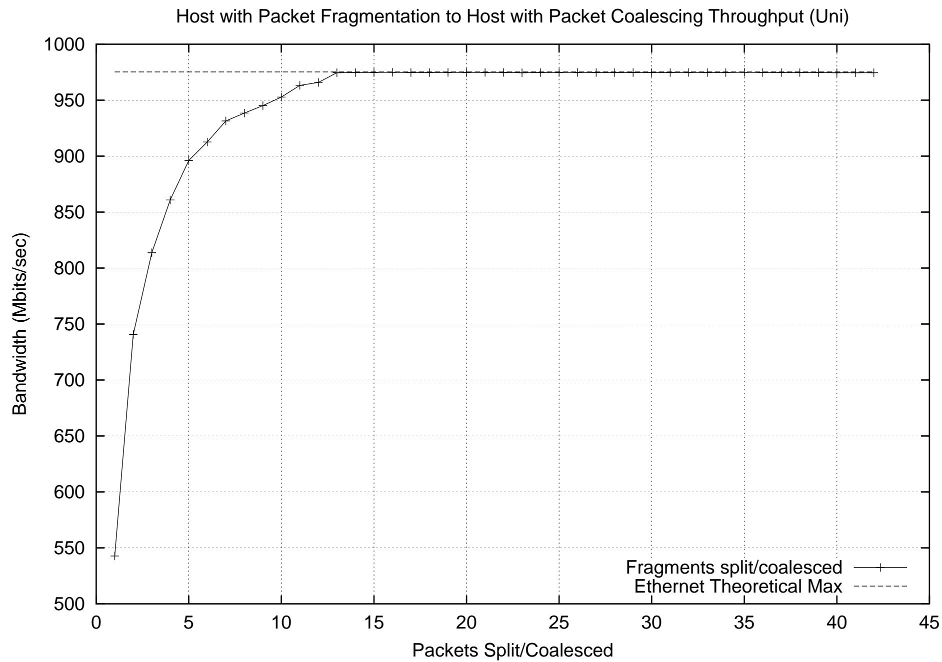
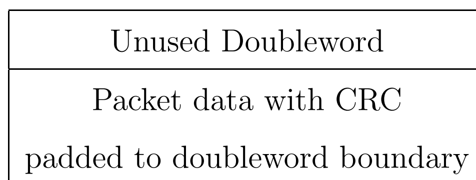


Figure 5.11: Host with Packet Fragmentation to Host with Packet Coalescing (using standard 1.5k packets throughout)

5.3.1 Pitfalls

When the MAC receives a frame from the network, it stores the packet to the on-board memory in the following format:



The host will thus end up receiving all combined packets in this format. Though this does not a problem in terms of processing, it is still not an ideal scenario for the host still has to waste some time to "defragment" the packets by stripping away the Ethernet headers. The SSRG is currently researching ways of avoiding this step.

Chapter 6

Conclusion

6.1 Results and achievements

This dissertation identifies the PCI as a major bottleneck. While packet-processing overheads are not insignificant, the performance loss is minimal when compared to the loss caused by moving small chunks of data from the hosts' memory onto the NIC (or vice-versa). Usually a lot of emphasis is placed on the maximum achievable bandwidth using the largest payloads possible. Unfortunately, small message transfers occur more frequently in every day applications. Indeed, even remote file systems, which are normally categorised as bulk transfer systems, depend heavily on the performance of small messages. A week-long trace of all NFS traffic on the departmental CS fileserver at UC Berkeley has shown that the vast majority of the messages is under 200 bytes in size and that these messages account for roughly half the bits sent [ACP95].

Furthermore, we have shown that the effect of extending the payload length not only creates compatibility problems, but also registers a considerable performance boost (at the physical line level). We have also shown that by utilising two rather simple techniques, performance at the application level could be boosted considerably without requiring radical changes in existing systems. Indeed, the whole operation is transparent from the rest of the networks' point of view.

6.2 Future work

Both *packet fragmentation* and *packet coalescing* techniques require host assistance thus involving some further host loading. The current implementation still requires the host to defragment packets by stripping away their headers if it needs to access the payload as a long stream of data. Some techniques, such as page remapping, have been developed which, if integrated with this system, would yield a further performance benefit. Better hardware support is needed for some of these operations to be offloaded to the NIC. For example, the embedded Alteon NIC CPUs only run at 66Mhz, which is not enough to maintain gigabit speeds if memory to memory copying on the NIC itself is involved.

6.3 Final remarks

In this dissertation we have identified key bottlenecks, we have experimented with different payloads and frame sizes, we have offloaded tasks to the NIC and improved throughput at the application level by a considerable 80%. However, we feel that while considerable progress has already achieved in shifting tasks traditionally reserved for the host onto the NIC, we believe that more research is required before the host ceases to be the major bottleneck.

Appendix A

Data Obtained

A.1 NIC to Host

Time (us)	Payload length	Packets sent	Throughput (Mbits/s)	PCI diff.
1506841	1498	69350	551.5449	114.0942
1197733	3010	34675	697.1287	107.5819
1032317	4522	23116	810.0655	43.0724
997948	6034	17337	838.6125	42.2689
966029	7546	13870	866.7485	42.9106
938764	9058	11558	892.1720	30.6817
925443	10570	9907	905.2269	26.7957
911958	12082	8668	918.6982	23.5970
905342	13594	7705	925.5443	22.3508
898472	15106	6935	932.7846	21.0338
892375	16618	6304	939.1556	20.2146
888580	18130	5779	943.2872	17.1445
883033	19642	5334	949.1870	15.1542
879771	21154	4953	952.7549	14.7454
877336	22666	4623	955.4827	13.8126
873950	24178	4334	959.2078	12.1783

874130	25690	4079	959.0291	14.5494
870288	27202	3852	963.1947	12.4632
869585	28714	3650	964.1942	13.2196
867305	30226	3467	966.6131	11.3230
866038	31738	3302	968.0765	10.8409
865542	33250	3152	968.6786	12.0457
863773	34762	3015	970.6942	10.7465
863695	36274	2889	970.6722	11.4916
861517	37786	2774	973.3376	9.7370
861034	39298	2668	974.1503	8.9244
860926	40810	2569	974.2151	9.3559
859046	42322	2476	975.8665	9.8385
859355	43834	2391	975.6815	10.1347
857150	45346	2311	978.0748	8.6468
857274	46858	2237	978.1829	8.8515
856461	48370	2167	979.0782	8.7120
855678	49882	2101	979.8273	8.7446
855215	51394	2039	980.2669	7.9943
854256	52906	1981	981.5024	7.3521
855328	54418	1926	980.2936	9.2822
853602	55930	1874	982.3109	7.3799
853195	57442	1825	982.9561	7.0590
853736	58954	1778	982.2260	8.3775
852329	60466	1733	983.5411	7.4913
852446	61978	1691	983.5677	7.8927
851862	63490	1651	984.4035	7.0220
851563	65002	1612	984.3849	7.2455

A.2 NIC to Host (with coalescing)

Fragments coal.	DMA length	Payload (bytes)	Bandwidth (Mbits/s)
1	1528	1498	530.8730
2	3056	3026	729.5356
3	4584	4554	810.2943
4	6112	6082	864.3500
5	7640	7610	895.3905
6	9168	9138	910.2290
7	10696	10666	928.9378
8	12224	12194	942.1483
9	13752	13722	947.3481
10	15280	15250	957.6750
11	16808	16778	964.3128
12	18336	18306	966.9150
13	19864	19834	973.4926
14	21392	21362	973.2046
15	22920	22890	973.7392
16	24448	24418	973.6922
17	25976	25946	975.1328
18	27504	27474	972.8670
19	29032	29002	974.2240
20	30560	30530	974.8811
21	32088	32058	974.1133
22	33616	33586	974.3269
23	35144	35114	974.4069
24	36672	36642	971.6794
25	38200	38170	974.9438
26	39728	39698	973.4572

27	41256	41226	973.2530
28	42784	42754	974.9948
29	44312	44282	974.2124
30	45840	45810	973.0523
31	47368	47338	972.5685
32	48896	48866	974.9424
33	50424	50394	974.4197
34	51952	51922	974.5686
35	53480	53450	975.0556
36	55008	54978	974.4631
37	56536	56506	974.5088
38	58064	58034	973.6479
39	59592	59562	974.2476
40	61120	61090	973.1490
41	62648	62618	973.5851
42	64176	64146	974.3659

A.3 Host to Host (standard firmware)

Time (us)	Payload (bytes)	Pkts. Transferred	Bandwidth (Mbits/s)
1544830	1498	69350	537.9818
1182456	3010	34675	706.1354
1033133	4522	23116	809.4257
985829	6034	17337	848.9217
958313	7546	13870	873.7272
938345	9058	11558	892.5703
925534	10570	9907	905.1379
914232	12082	8668	916.4131
907480	13594	7705	923.3638
898600	15106	6935	932.6518
893522	16618	6304	937.9500
894587	18130	5779	936.9532
890404	19642	5334	941.3294
886917	21154	4953	945.0784
888057	22666	4623	943.9477
884815	24178	4334	947.4293
888988	25690	4079	943.0004
883471	27202	3852	948.8221
882270	28714	3650	950.3313
880430	30226	3467	952.2033
882367	31738	3302	950.1613
879329	33250	3152	953.4907
880093	34762	3015	952.6941
878311	36274	2889	954.5192
877475	37786	2774	955.6362
877794	39298	2667	955.1924

877099	40810	2568	955.8791
877742	42322	2476	955.0804
876613	43834	2391	956.4731
875430	45346	2311	957.6515
876547	46858	2237	956.6752
875336	48370	2167	957.9662
874356	49882	2101	958.8962
875621	51394	2039	957.4221
874542	52906	1981	958.7353
873694	54418	1926	959.6867
874468	55930	1874	958.8716
873818	57442	1825	959.7573
873185	58954	1778	960.3483
873829	60466	1733	959.3417
872442	61978	1691	961.0248
873172	63490	1651	960.3788
872257	65002	1612	961.0307

A.4 NIC to Host PCI test

Time (us)	Bytes/transaction	Transactions	Bandwidth (Mbits/s)
6143465	128	819200	136.5452
4386357	192	546133	191.2430
3376714	256	409600	248.4252
2459962	384	273066	341.0048
2080587	512	204800	403.1847
1706780	768	136533	491.4862
1434085	1024	102400	584.9450
1260229	1512	69350	665.6390
1042434	3024	34675	804.7105
983233	4536	23116	853.1380
952266	6048	17337	880.8814
922167	7560	13870	909.6591
908956	9072	11558	922.8537
900027	10584	9907	932.0226
890151	12096	8668	942.2952
884905	13608	7705	947.8951
879473	15120	6935	953.8185
874308	16632	6304	959.3702
873392	18144	5779	960.4318
869776	19656	5334	964.3412
866936	21168	4953	967.5003
865368	22680	4623	969.2953
863493	24192	4334	971.3860
861536	25704	4079	973.5785
859613	27216	3852	975.6579
858242	28728	3650	977.4138

857660	30240	3467	977.9361
856825	31752	3302	978.9173
855271	33264	3152	980.7243
854659	34776	3015	981.4407
853919	36288	2889	982.1637
853300	37800	2774	983.0747
852771	39312	2667	983.5710
851595	40824	2568	984.8440
850753	42336	2476	985.7050
850792	43848	2391	985.8162
849901	45360	2311	986.7216
849840	46872	2237	987.0344
849153	48384	2167	987.7902
848347	49896	2101	988.5719
848528	51408	2039	988.2612
848129	52920	1981	988.8545
847523	54432	1926	989.5758
847449	55944	1874	989.6908
847318	57456	1825	990.0151
846717	58968	1778	990.6035
846082	60480	1733	991.0325
845851	61992	1691	991.4604
846015	63504	1651	991.4255
845523	65016	1612	991.6304

A.5 Host to NIC PCI test

Time (us)	Bytes/transaction	Transactions	Bandwidth (Mbits/s)
2913763	128	819200	287.8960
2014973	192	546133	416.3134
1703553	256	409600	492.4184
1383601	384	273066	606.2866
1239718	512	204800	676.6545
1094187	768	136533	766.6503
1025197	1024	102400	818.2435
940520	1512	69350	891.9083
880712	3024	34675	952.4766
848649	4536	23116	988.4339
833154	6048	17337	1006.8168
825607	7560	13870	1016.0495
815800	9072	11558	1028.2341
812367	10584	9907	1032.5943
809670	12096	8668	1035.9591
809529	13608	7705	1036.1545
804583	15120	6935	1042.5992
802795	16632	6304	1044.8309
805500	18144	5779	1041.3823
802625	19656	5334	1045.0221
801625	21168	4953	1046.3257
800625	22680	4623	1047.6779
801973	24192	4334	1045.9018
798459	25704	4079	1050.4897
797995	27216	3852	1050.9944
797732	28728	3650	1051.5532

798039	30240	3467	1050.9971
796416	31752	3302	1053.1692
796035	33264	3152	1053.7037
797456	34776	3015	1051.8413
796308	36288	2889	1053.2209
796142	37800	2774	1053.6532
795741	39312	2667	1054.0626
796543	40824	2568	1052.9102
794770	42336	2476	1055.1373
794777	43848	2391	1055.2954
795381	45360	2311	1054.3572
794884	46872	2237	1055.2751
794035	48384	2167	1056.3577
793610	49896	2101	1056.7558
794534	51408	2039	1055.4203
794081	52920	1981	1056.1595
794001	54432	1926	1056.2811
793818	55944	1874	1056.5551
794457	57456	1825	1055.8880
793401	58968	1778	1057.1714
793073	60480	1733	1057.2731
793048	61992	1691	1057.4742
793388	63504	1651	1057.1887
792540	65016	1612	1057.9231

A.6 Host Split to Host Coalesced

Time (us)	Packets transmitted	Bandwidth (Mbits/s)	Fragments split/coal
1105781	50001	542.7031	1
810132	50002	740.7716	2
737483	50001	813.7283	3
697217	50004	860.7748	4
669753	50005	896.0897	5
657591	50004	912.6445	6
644253	50001	931.4831	7
639550	50008	938.4643	8
635004	50004	945.1071	9
630001	50010	952.7267	10
629597	50006	953.2619	11
621389	50004	965.8150	12
615952	50011	974.4766	13
615773	50008	974.7014	14
615773	50010	974.7403	15
615786	50016	974.8368	16
615873	50014	974.6601	17
615778	50004	974.6155	18
615740	50008	974.7537	19
615903	50020	974.7295	20
615679	50001	974.7137	21
615742	50006	974.7116	22
615812	50002	974.5228	23
615884	50016	974.6816	24
615971	50025	974.7193	25
615968	50024	974.7046	26

615730	50004	974.6915	27
615791	50008	974.6729	28
616005	50025	974.6655	29
615822	50010	974.6628	30
615752	50003	974.6372	31
615827	50016	974.7719	32
616077	50028	974.6101	33
615906	50014	974.6079	34
615741	50015	974.8885	35
615776	50004	974.6187	36
616030	50024	974.6065	37
615847	50008	974.5843	38
616200	50037	974.5908	39
616252	50040	974.5670	40
616012	50020	974.5571	41
616041	50022	974.5502	42

A.7 Host Std. to Host Coalesced

Time (us)	Pkts recvd	Bandwidth (Mbits/s)	Pkts Coalesced
1105980	50001	541.7928	1
819898	50002	730.8518	2
749216	50001	799.7853	3
742125	50004	807.4757	4
742283	50005	807.32	5
742017	50005	807.6094	6
741517	50004	808.1378	7
741966	50001	807.6003	8
741451	50004	808.2098	9
741166	50005	808.5367	10
741048	50004	808.6493	11
740985	50001	808.6695	12
741079	50008	808.6801	13
741071	50004	808.6242	14
741044	50010	808.7506	15
741502	50006	808.1865	16
741529	50004	808.1247	17
741730	50011	808.0189	18
741705	50008	807.9976	19
740948	50010	808.8554	20
741109	50016	808.7768	21
741364	50014	808.4663	22
741621	50004	808.0245	23
740837	50008	808.9443	24
742003	50020	807.8669	25
741531	50001	808.0741	26

741571	50006	808.1113	27
742178	50002	807.3858	28
741309	50016	808.5586	29
741049	50025	808.9878	30
741901	50024	808.0426	31
740786	50004	808.9353	32
740885	50008	808.8919	33
741035	50025	809.0031	34
741589	50010	808.1563	35
742034	50003	807.5586	36
740956	50016	808.9438	37
741154	50028	808.9217	38
740881	50014	808.9933	39
741231	50015	808.6275	40
741722	50004	807.9145	41
741802	50024	808.1504	42

NB: The sender's limit is 810Mbits/s

A.8 Host to NIC

Payload (bytes)	Bandwidth (Mbits/s)
178	198.2333
242	263.1979
370	458.2524
498	532.2221
754	641.8731
1010	726.1357
1498	808.9748
3010	899.7057
4522	946.0275
6034	972.9891
7546	985.3899
9058	995.8060
10570	996.4021
12082	996.8564
13594	997.1970
15106	997.4704
16618	997.6984
18130	997.8942
19642	998.0601
21154	998.1880
22666	998.3062
24178	998.3915
25690	998.5202
27202	998.5820
28714	998.6515
30226	998.7379

31738	998.7503
33250	998.8605
34762	998.8734
36274	998.9327
37786	998.9936
39298	999.0100
40810	999.0111
42322	999.0833
43834	999.0752
45346	999.1527
46858	999.1981
48370	999.1510
49882	999.2401
51394	999.2180
52906	999.2409
54418	999.3099
55930	999.3222
57442	999.2461
58954	999.3071
60466	999.2489
61978	999.2930
63490	999.3459
65002	999.3187

A.9 Ethernet Theoretical Limits

True Frame size (bytes)	Real Payload	True Frames/s	True Payload bits/s	Theoretical Max Efficiency
88	50	1420454.55	568181818.2	568.1818
152	114	822368.42	750000000	750.0000
216	178	578703.70	824074074.1	824.0741
280	242	446428.57	864285714.3	864.2857
408	370	306372.55	906862745.1	906.8627
536	498	233208.96	929104477.6	929.1045
792	754	157828.28	952020202	952.0202
1048	1010	119274.81	963740458	963.7405
1536	1498	81380.21	975260416.7	975.2604
3048	3010	41010.50	987532808.4	987.5328
4560	4522	27412.28	991666666.7	991.6667
6072	6034	20586.30	993741765.5	993.7418
7584	7546	16482.07	994989451.5	994.9895
9096	9058	13742.30	995822339.5	995.8223
10608	10570	11783.56	996417797.9	996.4178
12120	12082	10313.53	996864686.5	996.8647
13632	13594	9169.60	997212441.3	997.2124
15144	15106	8254.09	997490755.4	997.4908
16656	16618	7504.80	997718539.9	997.7185
18168	18130	6880.23	997908410.4	997.9084
19680	19642	6351.63	998069105.7	998.0691
21192	21154	5898.45	998206870.5	998.2069
22704	22666	5505.64	998326286.1	998.3263
24216	24178	5161.88	998430789.6	998.4308
25728	25690	4858.52	998523010	998.5230

27240	27202	4588.84	998604992.7	998.6050
28752	28714	4347.52	998678352.8	998.6784
30264	30226	4130.32	998744382.8	998.7444
31776	31738	3933.79	998804128.9	998.8041
33288	33250	3755.11	998858447.5	998.8584
34800	34762	3591.95	998908046	998.9080
36312	36274	3442.39	998953514	998.9535
37824	37786	3304.78	998995346.9	998.9953
39336	39298	3177.75	999033963.8	999.0340
40848	40810	3060.13	999069721.9	999.0697
42360	42322	2950.90	999102927.3	999.1029
43872	43834	2849.20	999133843.9	999.1338
45384	45346	2754.27	999162700.5	999.1627
46896	46858	2665.47	999189696.3	999.1897
48408	48370	2582.22	999215005.8	999.2150
49920	49882	2504.01	999238782.1	999.2388
51432	51394	2430.39	999261160.4	999.2612
52944	52906	2360.99	999282260.5	999.2823
54456	54418	2295.43	999302188.9	999.3022
55968	55930	2233.42	999321040.6	999.3210
57480	57442	2174.67	999338900.5	999.3389
58992	58954	2118.93	999355844.9	999.3558
60504	60466	2065.98	999371942.4	999.3719
62016	61978	2015.61	999387254.9	999.3873
63528	63490	1967.64	999401838.6	999.4018
65040	65002	1921.89	999415744.2	999.4157

A.10 Host Split to NIC

Fragments	Payload length (bytes)	Bandwidth (Mbits/s)
1	1498	808.3237
2	2996	892.7797
3	4494	940.0449
4	5992	966.2615
5	7490	975.2454
6	8988	975.2455
7	10486	975.2118
8	11984	975.2454
9	13482	975.2448
10	14980	975.2461
11	16478	975.2453
12	17976	975.2454
13	19474	975.2455
14	20972	975.2454
15	22470	975.2452
16	23968	975.2455
17	25466	975.2308
18	26964	975.2454
19	28462	975.2461
20	29960	975.2454
21	31458	975.2459
22	32956	975.2453
23	34454	975.2460
24	35952	975.2234
25	37450	975.2400
26	38948	975.2378

27	40446	975.2374
28	41944	975.2456
29	43442	975.2455
30	44940	975.2457
31	46438	975.2454
32	47936	975.2304
33	49434	975.2282
34	50932	975.2276
35	52430	975.2434
36	53928	975.2440
37	55426	975.2423
38	56924	975.2415
39	58422	975.2017
40	59920	975.2142
41	61418	975.2098
42	62916	975.2349
43	64414	975.2344

A.11 NIC to NIC

Timer val	Frame Size (bytes)	Packets sent	Bandwidth (Mbits/s)
2709957	192	613078	568.8786
8769610	256	3033435	772.5793
14829265	384	5358235	907.0045
20888919	512	7140903	929.1458
26948571	768	8478011	952.1119
33008226	1024	9395515	963.8328
39067880	1512	10080084	975.4466
45127535	3024	10532455	987.5721
51187189	4536	10767229	991.6517
57246843	6048	10926443	993.7336
63306497	7560	11047042	994.9814
69366152	9072	11144150	995.8126
75425806	10584	11225445	996.4082
81485459	12096	11295364	996.8581
87545114	13608	11356705	997.2162
93604767	15120	11411344	997.4956
99664419	16632	11460604	997.7163
1.06E+08	18144	11505449	997.9038
1.12E+08	19656	11546607	998.0728
1.18E+08	21168	11584638	998.1908
1.24E+08	22680	11619983	998.3304
1.3E+08	24192	11652998	998.4314
1.36E+08	25704	11683971	998.5179
1.42E+08	27216	11713139	998.5914
1.48E+08	28728	11740702	998.6901
1.54E+08	30240	11766827	998.7481

1.6E+08	31752	11791656	998.7797
1.66E+08	33264	11815313	998.8496
1.72E+08	34776	11837903	998.8899
2746154	36288	212784	999.0145
8805807	37800	233504	998.9839
14865467	39312	253402	999.0512
20925114	40824	272539	999.0862
26984768	42336	290972	999.0910
33044422	43848	308750	999.0907
39104076	45360	325919	999.1416
45163730	46872	342519	999.1994
51223384	48384	358587	999.1804
57283038	49896	374155	999.2774
63342692	51408	389254	999.1972
69402346	52920	403911	999.2579
75462002	54432	418152	999.3658
81521653	55944	431998	999.3103
87581306	57456	445473	999.2979
93640960	58968	458594	999.3569
99700615	60480	471381	999.3149
1.06E+08	61992	483849	999.3635
1.12E+08	63504	496014	999.4020
1.18E+08	65016	507891	999.4141

Bibliography

[ACP95] T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations, 1995.

[BCF$^+$95] 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, 1995.

[BDHR95] T. Braun, C. Diot, A. Hoglander, and V. Roca. An Experimental User level implementation of TCP. *Technical Report, INRIA Sophia Antipolis, France*, (2650), 1995.

[CKS00] Felix Rauch Christian Kurmann, Michel Muller and Thomas Stricker. Speculative defragmentation — a technique to improve the communication software efficiency for gigabit ethernet. In *Proc. 9th IEEE Symp. High Performance Distr. Comp.*, pages 131–138, August 2000.

[CZ89] John. B. Carter and Willy Zwaenepoel. Optimistic implementation of bulk data transfer protocols. In *Proc. 1989 ACM SIGMETRICS and PERFORMANCE '89: International Conference on Measurement and Modeling of Computer Systems*, pages 61–69, Berkeley, CA, 23-26 1989. ACM Press.

[DRM$^+$98] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–??, /1998.

[GCY] Andrew Gallatin, Jeff Chase, and Ken Yocum. Trapeze/IP: TCP/IP at near-gigabit speeds. pages 109–120.

[Gro] PCI Special Interest Group. Pci local bus specification.

[KCT98] Y. Miller K. Claffy and K. Thompson. The nature of the beast: Recent trac measurements from an internet backbone, 1998.

[Pos81] Jon B. Postel. Transmission Control Protocol. Technical Report 793, SRI International, 1981.

[STH+99] S. Sumimoto, H. Tezuka, A. Hori, H. Harada, T. Takahashi, and Y. Ishikawa. The design and evaluation of high performance communication using a gigabit ethernet, 1999.

[TOHI] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. pages 308–315.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface of parallel and distributed computing, 1995.

[Weba] Alteon Websystems. Extended frame sizes for next generation ethernets - a white paper.

[Webb] Alteon Websystems. Next generation adapter design and optimization for gigabit ethernet.

[Webc] Alteon Websystems. Tigon/pci ethernet controller documentation.