The Performance of a Linux NFS Implementation

by

Christopher M. Boumenot

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

____

May 20, 2002

APPROVED:

____

Prof. David Cyganski, Major Advisor

____

Prof. John Orr

____

Prof. Donald R. Brown

Abstract

NFS is the dominant network file system used to share files between UNIX-derived operating system based hosts. At the onset of this research it was found that the tested NFS implementations did not achieve data writing throughput across a Gigabit Ethernet LAN commensurate with throughput achieved with the same hosts and network for packet streams generated without NFS. A series of tests were conducted involving variation of many system parameters directed towards identification of the bottleneck responsible for the large throughput ratio between non-NFS and NFS data transfers for high speed networks. Ultimately it was found that processor, disk, and network performance are not the source of low NFS throughput but rather it is caused by an avoidable NFS behavior, the effects of which worsen with increasing network latency.

Acknowledgements

First and foremost I would like to thank Professor Cyganski who started this project. Without him, I would not have had such an interesting thesis and enjoyable graduate career. Professor Cyganski has constantly been there to support and guide me from the beginning of this project until its conclusion. He has been there during the ups and downs that go along with a thesis as well as the dead ends, wrong ways, U-turns, and other miscellaneous road signs experienced along the way. For all of his help, I am deeply indebted.

I would like to thank Professor Brown for being a member of my committee, a good friend, and a professor. He taught me that with probability and linear systems theory you can do an ything in Electrical Engineering.

Professor Orr is the head of the ECE department, and in my six year career at WPI, I have not had the chance to meet with Professor Orr more than a few times. I am nonetheless appreciative of the work he has done that has affected me both directly and indirectly. Furthermore, I would like to thank him for accepting a position on my committee, and lending his thoughts and ideas to improve this thesis.

I would like to thank Pirus Networks for funding this thesis. I would also like to thank them for allowing us to develop a worthwhile thesis. Professor Cyganski and I determined in the middle of the project that the largest benefit we could do for Pirus and this thesis was by changing our originally stated intentions. Without hesitation, Pirus allowed us to change directions and explore that alternative.

Contents

1 Introduction **1**
1.1 NFS 2
1.2 Thesis Organization 3

2 Network Benchmarks **4**
2.1 Network Configuration 4
2.2 Netperf 5
2.3 Loopback 8
2.4 Network Throughput 8
2.5 Fast Ethernet 10
2.6 Gigabit Ethernet 11
2.6.1 Jumbo Frames 13
2.6.2 Burst Mode 13
2.6.3 Zero-Copy 14

3 Introduction to NFS **15**
3.1 NFS Protocols 15
3.2 NFS 16
3.3 NFS Versions 18
3.4 Default Testing Conditions 19

4 NFS Benchmarks **20**
4.1 Connectathon 21
4.2 Multiple Tasks 25
4.3 Loopback 30
4.4 Network Latency 33

5 An Interim Performance Model **36**
5.1 Processor Bound Performance Model 36
5.2 MTU Variation 37

6 An Interim Performance Model Refutation **40**
6.1 Disk Speed 40
6.2 FreeBSD 46

7 Round Trip Time Benchmarks **49**
7.1 UDP and RPC RTT 49

8 IP Fragments **53**
8.1 IP Fragmentation 53
8.2 Linux IP Fragments 55
8.3 Variable Adjustment 55
8.4 Testing for IP Fragments 56
8.5 MTU's Effect on Performance 58

9 A Transactional Model **61**
9.1 Core Processor Speed 61
9.2 Network Latency 63
9.3 Disk Latency 64
9.4 Model of NFS Operations 65
9.4.1 Loopback NFS Latency Model 68
9.5 Calculations 69
9.5.1 Host-to-Host 69
9.5.2 One NFSD 71
9.5.3 Loopback 72
9.5.4 Revisiting Host-to-Host Throughput 73
9.6 NFS vs. Processor Load 74
9.7 NFS's Network Pace 74

10 Concurrency Observed **77**
10.1 Multiple NFS Servers and Clients 77
10.2 Multiple NFS Servers 79
10.3 Multiple NFS Clients 80

11 NFS Packet Trace Analysis **82**
11.1 Fast Disk Packet Trace 82
11.2 Slow Disk Packet Trace 86

12 Conclusion **89**
12.1 Black Box Model 90
12.2 Future Work 90

A Inventory **93**
A.1 NFS Test Hosts 93

B Source Code **95**
B.1 Parse Connecathon Log Files 95
B.2 Connectathon Write Shell Scripts 97
B.3 Threaded Connectathon Write Test 99
B.4 Ethereal NFS Packet Analyzer 101
B.5 Parse Linux's /proc 105
B.6 UDP RTT Client 108
B.7 UDP RTT Server 109
B.8 Precise Timer Class Definition 110
B.9 Precise Timer Class Implementation 112
B.10 UDP Socket Class Definition 114
B.11 UDP Socket Class Implementation 116
B.12 UDP RTT Makefile 118
B.13 Compute Average from UDP RTT Output 119
B.14 Increase Processor Load 120

C Default Operating System Parameters **122**
C.1 Linux Parameters 122
C.1.1 TCP/IP 122

D NFS Tests **124**
D.1 Additional NFS Tests 124

List of Figures

2.1 Loopback diagram. 9
2.2 Linear relationship between processor speed and throughput in loopback. 10

4.1 Polaris+ to Hutt at 1 Gbps. 23
4.2 Legacy to Hutt at 100 Mbps. 24
4.3 Polaris+ to Hutt at 1 Gbps with multiple tasks, and 1MB file size 26
4.4 Linear relationship as the number of tasks is increased 27
4.5 Polaris+ to Hutt at 1 Gbps with multiple tasks, and 1MB file size, and different files per each write sub-test. 28
4.6 Polaris+ to Hutt at 1 Gbps with multiple tasks, and 32MB file size. 29
4.7 Polaris+ Loopback with multiple tasks, and 1MB file size . . . 31
4.8 Polaris+ Loopback with multiple tasks, and 1MB file size . . . 32
4.9 Topology used for NFS network testing. 34
4.10 Alternative topology used to test network latency's effect. . . . 34
4.11 Polaris+ to Hutt at 1 Gbps through alternate topology. 35

5.1 Packet inter-arrival times compared to packet processing times. 37

6.1 Revanche to Beast at 1Gbps, multiple tasks, 1MB file size, and disk with 2.89 MB/sec transfer speed. 41
6.2 Revanche to Beast at 1Gbps, multiple tasks, 32MB file size, very slow disk. 42
6.3 Revanche to Bastion at 1Gbps, multiple tasks, 1MB file size, slow disk. 42
6.4 Revanche to Bastion at 1Gbps, multiple tasks, 32MB file size, slow disk. 43

6.5 Revanche to Beast at 1Gbps, multiple tasks, 1MB file size, fast disk. 44
6.6 Revanche to Beast at 1Gbps, multiple tasks, 32MB file size, fast disk. 44
6.7 FreeBSD: Kaboom to Crash at 100 Mbps. 47
6.8 (Linux) Kaboom to Crash at 100 Mbps. 47

7.1 Near-linear relationship between packet size and UDP RTT. . 51
7.2 Near-linear relationship between packet size and RPC RTT. . 52

8.1 Loopback performance of Revanche for varying MTU sizes. . . 59
8.2 Beast to Revanche performance for varying MTU sizes connected directly by a crossover cable. 59

9.1 Loopback and host-to-host performance for different MTU sizes for Revanche and Beast. 62
9.2 NFS model of a transaction. 65
9.3 NFS latency diagram for sequential transfer with a host-to-host configuration. 66
9.4 NFS latency diagram for two concurrent transfers with a host-to-host configuration. 66
9.5 NFS latency diagram, loopback mode for sequential transfers. 68
9.6 NFS latency diagram, loopback mode for concurrent transfers. 69
9.7 Processor load's effect on NFS performance. 75

11.1 This plot shows the cumulative transfer versus time for a Connectathon write test between Revanche and Beast. 84
11.2 Per transaction throughput for write test described in Section 11.1. 85
11.3 The time between write call requests and write call replies. . . 85
11.4 This plot shows the cumulative transfer versus time for a Connectathon write test between Beast and Revanche (slower hard disk). 87
11.5 Per transaction throughput for test described in Section 11.1. . 88
11.6 The time between write call requests and write call replies. . . 88

D.1 Polaris+ to Hutt, varying block size 124

List of Tables

2.1 NIC information. 5
2.2 Netperf benchmarks. 5
2.3 Netperf TCP test results in loopback. 8
2.4 Netperf UDP test results in loopback. 9
2.5 TCP Netperf results between Beast and Revanche. 11
2.6 UDP Netperf results between Beast and Revanche. 12

4.1 Average multitasked throughput between Polaris+ and Hutt with 1MB Files. 27
4.2 Polaris+ to Hutt at 1 Gbps with multiple tasks, and 1MB file size, and different files per each write sub-test. 29
4.3 Polaris+ to Hutt at 1 Gbps with multiple tasks, and 32MB file size. 30
4.4 Polaris+ Loopback with multiple tasks, and 1MB file size. . . 31
4.5 Polaris+ Loopback with multiple tasks, and 32MB file size. . . 32

5.1 Connectathon write test on Polaris+ through loopback. 38
5.2 Connectathon write test, Polaris+ to Hutt at 1 Gbps. 38

6.1 Throughput with 32MB files, and multiple tasks for different disk speeds. 45

7.1 UDP round trip times between Polaris+ and Hutt. 50
7.2 RPC round trip times between Polaris+ and Hutt. 50
7.3 UDP throughput with FTP-like transfers. 51

8.1 IP fragmentation results for direct crossover connection. 57
8.2 IP fragmentation results when connected through a Gigabit Ethernet Switch. 57
8.3 Jumbo frame performance through a direct crossover connection. 58

9.1 Comparison of hdparm and Connectathon measured write throughput. 65
9.2 Host-to-host latency NFS performance calculations between Revanche and Beast based on $T_{op} = 0$ approximation. 71
9.3 Effect of different network latencies and disk speeds on performance with a single nfsd. 72
9.4 host-to-host latency NFS performance calculations for a single nfsd. 74
9.5 Packet trace of a Connectathon write test. 76

10.1 Packet trace of Connectathon write test between Revanche and Beast. 78
10.2 NFS write test with Hutt as the client, and Beast and Revanche as the servers. 79
10.3 NFS write test executed separately between Hutt as the client and Revanche as the server, and Hutt as the client and Beast as the server. 79
10.4 NFS write test with Hutt as the server, and Beast and Revanche as the clients. 80

11.1 NFS RPC write call retransmissions, and duplicates. 83

Chapter 1

Introduction

If one measures the speed of NFS (Network File System) write bandwidth in a loopback system, that is, one in which a local disk is NFS mounted hence incurring all network protocol costs but avoiding the physical layer one can obtain performance as high as 762 Mbps. Given that measurements of a Gigabit network between two machines yields throughput as high as 990 Mbps, it is conceivable that host-to-host NFS performance could equal loopback performance. However, our measurements show that performance may be as low as 61 Mbps. This observation gives rise to three goals of this thesis.

The first goal is to discover the bottlenecks that give rise to this loss of performance. This was accomplished by using industry standard benchmarks, and tools we developed to assess specific behaviors of NFS and the network. We observed the impact of results unbiased by NFS caching, under variations of network latency, disk speed, varying degrees of host performance, network round trip times, and network throughput.

Our second goal is to obtain a model for NFS performance that explains the above performance values in terms of component performance parameters. The models developed allowed us to assign a metric to the performance of a system's TCP/IP stack, datagram processing based upon MTU size, expected loopback mode throughput, and expected NFS performance.

Our third goal is to develop a suite of tests that extract the performance parameters needed to populate the performance model by direct measurement of an otherwise black box NFS system. A model can be constructed to put an upper bound on the performance of NFS assuming no caching is involved. The parameters needed for this model are network propagation

delay, the NFS block size, the throughput of the hard disk, or RAID array, and the throughput of an NFS system in local loopback mode.

1.1 NFS

At the most basic level a filesystem is used to reliably store and retrieve a user's data. Filesystems usually implement a hierarchy so that data may be maintained in a much more ordered fashion. This is usually done using files and directories to store and organize the data.

A filesystem must be reliable, transparent, and maintain data integrity. NFS (Network File System) is a type of filesystem that stores user's data on a machine other than the user's local one. The user is able to make a network connection to communicate with the filesystem transparently using the NFS protocol. This means that instead of having the data stored on one's personal machine it is instead stored on the network.

From an administrative point of view it is difficult to maintain, organize, and protect data for many independent users. Bringing the user's data into one central location makes these three requirements easy to implement. But, measures implemented for the convenience of the administrator must be made transparent to user in order to not disturb their working habits.

NFS gives each user the ability to manipulate their data as if it was stored locally on their machine, but at the same time allowing the data to actually reside in a different, more convenient location for the administrator.

A problem with NFS is that performance can suffer due to the fact that data must be moved across the network. The user is at the mercy of the performance of the network, and must deal with network issues such as congestion, and down time.

A second problem is a potentially wide variety of clients running different operating systems all want access to data on the server. Compared to a locally mounted filesystem, a network file system has more incompatibility issues to address. NFS does not have the luxury of servicing only one machine, one user, and one operating system at a time.

An NFS server works by *exporting* a filesystem to clients. The clients can then *mount* the exported filesystem as if it was a local one. The communication between the NFS client and server is handled by RPC (Remote Procedure Call) which runs on top of a network protocol, either UDP, or TCP. All of these things together ensure that a consistent view of the filesys-

tem is maintained across all exported filesystems and users, and that data on an NFS server is treated no differently from that on a user's machine.

1.2 Thesis Organization

This thesis is more like a detective story than a technical paper in that piece by piece evidence is gathered and discarded. The evidence is assessed, reasons given for its nature and conclusions that may be drawn from it are discussed. Finally, after careful review of all of the evidence an accurate model to describe NFS write performance is developed and verified.

Chapter 2 presents results for the network used to test NFS performance. Chapter 3 gives a brief introduction to NFS. NFS performance is investigated in Chapter 4 with the use of benchmarks to uncover performance under different conditions. Scenarios are developed to help explain performance in Chapter 5, and refuted in Chapter 6. Other possible NFS bottlenecks are examined in Chapter 8. In Chapter 9 an accurate model of NFS is described. This model is able to predict the upper bound on NFS performance within 2% of test results. The paper is concluded in Chapter 12.

Chapter 2

Network Benchmarks

The first step in the process of reviewing NFS performance was to test and quantify the performance of an Ethernet network. Specifically, we were concerned with Gigabit Ethernet over Fiber and Copper, and Fast Ethernet. The performance of the network was measured using many different benchmarks. Some of the benchmarks used were industry standard benchmarks such as Netperf (as described in Section 2.2), and others were developed to test performance of specific protocols. The network protocol related performance tests included TCP, UDP, and RPC throughput, and file transfer times using an FTP-like transfer mechanism.

2.1 Network Configuration

The machines used to test network performance were equipped with Gigabit Ethernet, and Fast Ethernet adapters. All of the network cards used were manufactured by Intel. Four Gigabit Ethernet adapters were used, two with copper connectivity and the remaining ones with fiber connectivity. The Fast Ethernet adapters have only copper connectivity. The Gigabit Ethernet adapters with only copper connectivity were connected with a crossover cable. The remaining adapters were connected through the ECE department's switch, which has a 40 Gbps backplane. The Gigabit Ethernet adapters were connected to the processor via a 64-bit/66MHz PCI bus for the machines Beast and Revanche, but all other adapters were connected via a 32-bit/33MHz PCI bus. Table 2.1 displays a listing of the adapters used, manufacturer, model numbers, physical connection, and driver version.

Manufacturer	Model	Connection Type	Driver Version
Intel	PRO/1000XT	Copper	Intel 4.1.7
Intel	PRO/1000F	Fiber	Intel 4.1.7
Intel	PRO/100	Copper	Linux 2.4.18

Table 2.1: NIC information.

2.2 Netperf

The Netperf benchmark was used to measure the performance of Ethernet LAN connections with 100 Mbps, and 1 Gbps link speeds. The Netperf benchmark is able to test different aspects of a network. Those tests that are relevant to our experiments are listed in the Table 2.2.

Test Name	Description
TCP_STREAM	TCP stream throughput
UDP_STREAM	UDP stream throughput

Table 2.2: Netperf benchmarks.

The Netperf benchmark was developed by Rick Jones of Hewlett-Packard, and is available from **http://www.netperf.org**. It is capable of measuring TCP and UDP stream performance, TCP request, response performance, and TCP connect, request, and response performance. Netperf also has support for DLPI, XTI, and ATM. The test works by running a daemon on one host (server), and a client on another host. The client program has command line options that can be used to select the test to be run, and to change the default behavior of the test. For example, the send and receive buffers can be changed. For a complete list of options see the Netperf manual page.

The benchmarks executed involved using machines all the way from a Pentium II, 300 MHz, up to a Pentium IV, 1.7 GHz server. (Characteristics of the machines used can be found in Appendix A, and are keyed by machine name.) Netperf was launched on the client with only two command line options, one to specify the server's IP address, and one for the length of the test. The Netperf documentation recommends running all tests for a period of at least one minute, and running enough trials to ensure results are consistent. The results presented reflect a test time of one minute, and

a total of three trials.

All of the tests were run again but with modifications to the operating systems default TCP parameters. From researching the USENET archives we found suggestions indicating modifying tcp_mem, tcp_wmem, tcp_rmem, and MTU (Maximum Transmission Unit) would have a large effect on network performance. The USENET archives can be browsed and searched at **http://groups.google.com**.

The following information was taken from the standard kernel distribution of Linux as found on **http://kernel.org**, in the Documentation directory[1]of the kernel. This documentation was found in kernel 2.4.17, but can be found in future revisions as well as previous versions.

```
tcp_mem - vector of 3 INTEGERs: min, pressure, max

    low: below this number of pages TCP is not bothered
    about its memory appetite.

    pressure: when amount of memory allocated by TCP
    exceeds this number of pages, TCP moderates its
    memory consumption and enters memory pressure mode,
    which is exited when memory consumption falls under
    "low".

    high: number of pages allowed for queueing by all TCP
    sockets.

    Defaults are calculated at boot time from amount of
    available memory.

tcp_wmem - vector of 3 INTEGERs: min, default, max

    min: Amount of memory reserved for send buffers for TCP
    socket. Each TCP socket has rights to use it due to fact
    of its birth. Default: 4K

    default: Amount of memory allowed for send buffers for
```

[1] ./Documentation/networking

```
    TCP socket by default. This value overrides
    net.core.wmem_default used by other protocols, it is
    usually lower than net.core.wmem_default. Default: 16K

    max: Maximal amount of memory allowed for automatically
    selected send buffers for TCP socket. This value does not
    override net.core.wmem_max, "static" selection via
    SO_SNDBUF does not use this.  Default: 128K

tcp_rmem - vector of 3 INTEGERs: min, default, max

    min: Minimal size of receive buffer used by TCP sockets.
    It is guaranteed to each TCP socket, even under moderate
    memory pressure.  Default: 8K

    default: default size of receive buffer used by TCP
    sockets.  This value overrides net.core.rmem_default used
    by other protocols. Default: 87380 bytes. This value
    results in window of 65535 with default setting of
    tcp_adv_win_scale and TCP_app_win:0 and a bit less for
    default tcp_app_win. See below about these variables.

    max: maximal size of receive buffer allowed for
    automatically selected receiver buffers for TCP socket.
    This value does not override net.core.rmem_max, "static"
    selection via SO_RCVBUF does not use this.  Default:
    87380*2 bytes.
```

MTU specifies the maximum transmission unit used by IP when assembling a datagram to accommodate the largest packet an interface or intervening network is capable of supporting. In the case of Ethernet the maximum size is 1,500 (expect for the special case with Gigabit Ethernet when used with jumbo frames). RFC 1122 specifies that any node on the Internet must be capable of accepting packets of at least 576 bytes in size. The size specified for an MTU can be either manually set or determined by the PATH MTU discovery algorithm when implemented. The PATH MTU discovery algorithm is defined in RFC 1191.

RFC 791 describes the operation a node must perform when a packet is

received which is larger than the network node is capable of handling. The receiving network must fragment the packet on octet (8-byte) boundaries. A source node can request a packet not be fragmented by marking the packet. If a node must fragment a packet marked as such, it is instead dropped, and an ICMP error message is sent back to host.

2.3 Loopback

Loopback is a mode of operation in which a node acts as both a client and a server. In this mode the node's TCP/IP stack is traversed twice, and the NIC and NIC driver is not used. A graphic depicting this can be seen in Fig. 2.1. All layers above the dashed line marked loopback represent the layers present in loopback. The data link and physical layer would not be present for a node in loopback mode.

2.4 Network Throughput

Performance evaluation in loopback mode was chosen as a starting point because one can see the raw performance of the machine without network bottlenecks. Loopback roughly shows half the performance that might be expected from a particular machine because the performance is processor speed bound (that is, there is no idle time during prosecution). In loopback, the host must act as both the sender and receiver. Netperf was executed on three different machines including Bastion, Hutt, and Revanche. The baseline TCP and UDP performance results are summarized in Table 2.3, and for UDP in Table 2.4.

Host	Processor	Processor Speed (MHz)	Average Throughput (Mbps)	Standard Deviation (Mbps)
Bastion	PII	350	628.21	10.96
Hutt	Quad PII Xeon	700	1854.14	0.71
Revanche	Athlon XP	1553	4299.46	70.9126

Table 2.3: Netperf TCP test results in loopback.

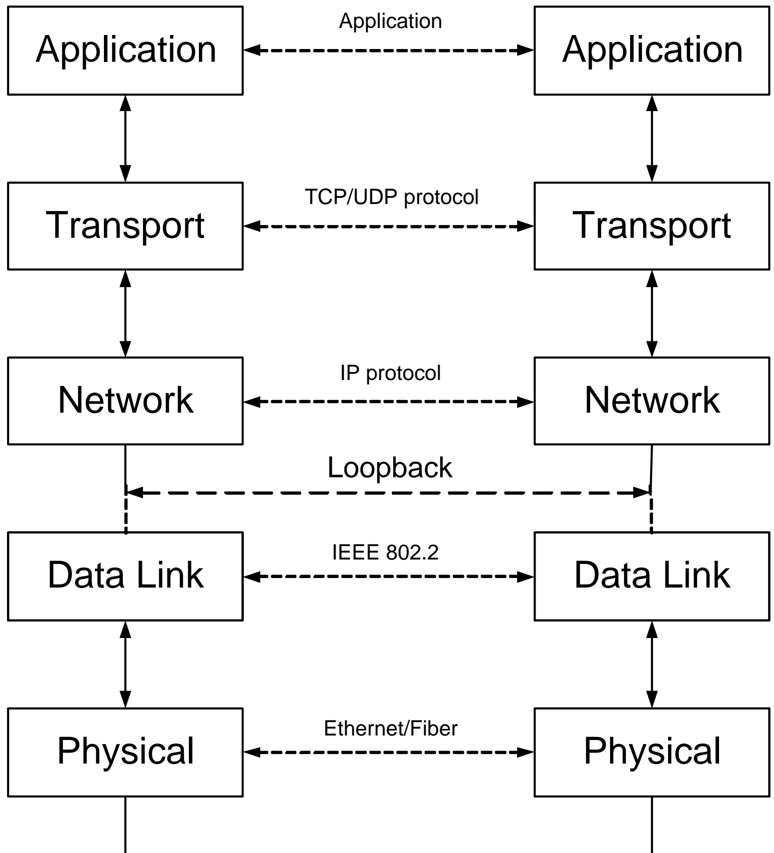

Figure 2.1: Loopback diagram.

Host	Processor	Speed	Average (Mbps)	σ
Bastion	PII	350 MHz	437.33	8.44
Hutt	PII Xeon	700 MHz	1439.08	2.18
Revanche	Athlon XP	1553 MHz	4674.81	51.48

Table 2.4: Netperf UDP test results in loopback.

The results show that our machines would not be processor limited on a connection through a network. A PII, 350 MHz machine is able to achieve approximately 630 Mbps in loopback. Because loopback requires twice the processing power, this machine should be able to achieve approximately 1.2 Gbps when coupled with a machine of equal or greater processing power. A near linear performance is observed compared to processor speed which indicates that in loopback mode, NFS throughput is exactly processor bound. This linear relationship is plotted in Fig. 2.2.

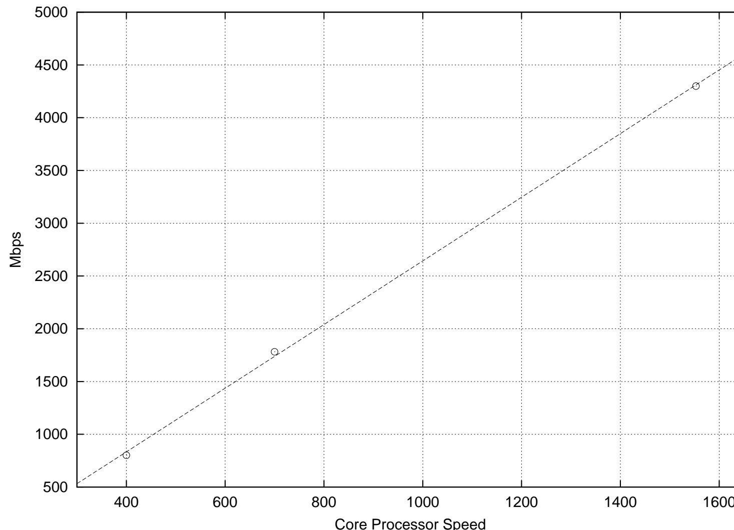

Figure 2.2: Linear relationship between processor speed and throughput in loopback.

2.5 Fast Ethernet

The second test involved running Netperf between Legacy and Polaris over a 100 Mbps connection with the default TCP options of Linux kernel version 2.4.16. The average Netperf measured results of the three trials was equal to 94.093 Mbps, with a standard deviation of 0.03. The second half of this test involved the modification of tcp_wmem, and tcp_rmem to a value of '65536 65536 65536', this lead to a throughput of 94.12 Mbps with a standard

deviation of zero. This results translates into 99.1% of the theoretical maximum performance in Mbps that can be achieved using Ethernet. Clearly we obtained network limited performance in this case.

The theoretical maximum performance of Ethernet was calculated by assuming a standard TCP/IP header. A standard Ethernet frame is 1500 bytes in size. The IP header consumes 20 bytes, and the TCP header consumes 20 bytes, consuming a total of 40 bytes. Ethernet has a 8 byte preamble, 6 byte destination and source address, 2 bytes length or type field, 4 bytes for frame check sum (FCS), and 12 bytes used for inter frame spacing [2].

$$94.93\% = \frac{1460}{8 + 6 + 6 + 2 + 1500 + 4} \cdot 100 \tag{2.1}$$

2.6 Gigabit Ethernet

The primary network connection to be used for testing in this thesis is Gigabit Ethernet with both switched and crossover connectivity. The machines used in this test were Beast and Revanche both running Debian Linux v3.0 with kernel version 2.4.18. The results presented in this section present baseline results as well as performance upon modification to Linux's TCP/IP parameters. The baseline test revealed a Netperf measured maximum throughput of 941.29 Mbps. The parameters tcp_wmem, and tcp_rmem were set to '1048576 1048576 2097152', and the MTU was adjusted to 12,000 bytes. This yielded a result of 990.00 Mbps, with a standard deviation of 0. These results are also presented in Table 2.5.

MTU	Connection	tcp_wmem/tcp_rmem (bytes)	Throughput (Mbps)
1,500	Switched	default[3]	470.22
1,500	Switched	'1048576 1048576 2097152'	760.10
1,500	Crossover	default	941.29
12,000	Crossover	'1048576 1048576 2097152'	990.00

Table 2.5: TCP Netperf results between Beast and Revanche.

[2] *Gigabit Ethernet*, Rich Seifert, pg 169, 177

[3] Default values for operating system specific variables are listed in Appendix C.

The results shown in Table 2.5 illustrate that the default parameter's performance of crossover connectivity compared to the performance as a result of tuning the default parameters is small. A 1% increase in performance was seen as a result of tuning parameters for crossover connectivity. The parameters had to be tuned much more aggressively to yield that 1% increase in performance, as opposed to Fast Ethernet. An MTU value larger than 12,000 bytes was not considered because Ethernet's 32-bit CRC is not effective above an MTU of 12,000 bytes.

The performance of UDP was also tested with Netperf for both a switched, and crossover connection. The testing strategy recommended by Netperf for UDP is the same as TCP except that the throughput of the receiver should be used over the throughput of the sender.

A Netperf UDP stream test is composed of two parts. The first part involves the client transmitting to the server, and a measurement of the bandwidth achieved is recorded. The second part involves the server transmitting to the client, and a second measurement of the bandwidth achieved is recorded. The results of these tests are presented in Table 2.6.

MTU	Connection	Send Throughput (Mbps)	Receive Throughput (Mbps)
1,500	Switched	748.69	451.96
1,500	Crossover	825.09	824.61
12,000	Crossover	868.02	868.02

Table 2.6: UDP Netperf results between Beast and Revanche.

There is a large discrepancy, 65%, between the performance of send and receive for UDP over a switched network. It is important to note the difference between send and receive throughput performance. The measurement for both values is recorded on the client, and not on the server therefore; the send performance does not take into account lost UDP packets whereas receive performance does. The reason for packet loss is because UDP does not implement flow control thereby allowing a sender to easily overcome the receiver's ability to properly buffer packets. As packets are being lost UDP is further hampered because it does not react to this information, and will continue to drown the receiver with UDP packets.

The loss of UDP packets due to buffer overflow is recorded in Linux as SNMP information. This information can be found in snmp under the proc

filesystem (/proc/net/snmp). We observed the values reported by Linux before and after the test, and found that 1,489,417 UDP packets were lost as the result of a 60 second Netperf UDP stream test. Later, it will be shown that this packet loss is not responsible for low NFS throughput performance.

We attribute the performance difference between a switched and crossover connection as being the result of the switch's inability to buffer packets, and UDP's inability to react to adverse conditions in the network.

2.6.1 Jumbo Frames

Jumbo frames were used in some tests to increase the throughput for Gigabit connections by making the fragmentation of IP packets less frequent with the larger size. Jumbo frames typically allow an Ethernet frame size of 9,000 bytes, but can be made larger or smaller depending upon the manufacturer of the NIC. Support for a frame size of 9,000 is popular because of NFS's 8 KB block size. An MTU size of 9,180 is recommended by RFC 1626 for ATM AAL5 supporting IP over ATM. IETF's Network Working Group's Internet draft "Extended Ethernet Frame Size Support[4]," recommends an MTU of 4,470 to support future applications that will make use of extended ethernet frames, and to prevent fragmentation of FDDI.

The support of jumbo frames between different hosts across a network is determined by using VLAN tagging[1] as defined in IEEE 802.1Q. Nodes can be placed on their own network segment to avoid VLAN requirements. In our configuration hosts were placed on a separate network segment.

2.6.2 Burst Mode

Gigabit Ethernet has the ability to operate in burst mode which allows it to transmit a series of frames up to a max of 64KB without releasing the transmission medium. The advantage is for non-fiber Ethernet (CDMA/CD) which must constantly sense the line to determine if it has the ability to transmit. In this case burst mode provides a means to bypass the bandwidth wasting inter-frame sense time. On the other hand, fragmentation of large datagrams into a 1,500 byte MTU is still required in burst mode since burst mode does not augment a node's MTU.

[4]http://www.ietf.org/internet-drafts/draft-ietf-isis-ext-eth-01.txt

2.6.3 Zero-Copy

When a packet arrives at a host it is the responsibility of the kernel to copy the packet from the NIC into its own address space. The packet must again be copied from the kernel's address space to that of the user's address space. The penalty incurred for having to copy the buffer multiple times is large, and can greatly affect network communication performance.

Providing means for a packet to be copied directly into a user's address space, and bypassing the kernel is called zero-copy support. Zero-copy can improve network communication performance considerably. Khalidi[13] found a 40% increase in network communications throughput, and a 20% decrease in CPU utilization as a results of zero-copy. Zero-copy support is complex to implement, and most operating systems, and NICs do not provide the functionality to make zero-copy possible. The complexity in zero-copy stems from the fact that two sources of information are needed to copy a packet to a NIC. The kernel provides the TCP/IP header information, and the user's process provides the actual data. The technique of bringing the two separate source of information together into one stream is called scatter-gather.

Zero-copy was introduced into the Linux kernel early in the 2.4 series by David S. Miller. Not all areas of the kernel make use of zero-copy, notably NFS, but there are patches available to add this functionality to NFS.

Additional techniques exist to improve network efficiency between the NIC and the operating system. They involve the more efficient movement of data between the NIC and the operating system. The Transport Area working group of the IETF is working on a draft for these techniques. Their work is available at **http://www.ietf.org/**[3].

Discovery of the fact that Linux's NFS implementation[4] did not support zero-copy, and that kernel patches were available to add support, happened after a majority of the results were collected. A test was conducted to observe the impact of NFS with zero-copy between Beast and Revanche, and there was no appreciable improvement in results found hence zero-copy is not used. The reason for this is made evident in Chapter 9.

[3]http://www.ietf.org/internet-drafts/draft-ietf-isis-ext-eth-01.txt, http://www.ietf.org/proceedings/00jul/I-D/kaplan-isis-ext-eth-02.txt

[4]Hirokazu Takahashi of VA Linux, NFS zero-copy patch developer, http://www.geocrawler.com/lists/3/SourceForge/789/50/8370662/

Chapter 3

Introduction to NFS

Network File System (NFS) is a stateless distributed file system that was invented by Sun Microsystems, and placed into the public domain in 1989 (RFC 1094). An NFS server exports a native file system so multiple clients can remotely mount said file system. The NFS server presents a consistent view to all clients, and manages client operations in a consistent manner.

3.1 NFS Protocols

NFS is built upon a layered protocol approach, much like other communication protocols that conform to the spirit of the OSI model. NFS is the top layer sitting upon XDR, RPC, TCP or UDP, IP, link and physical layers such as IEEE 802.2 and Ethernet. Each layer in the system has a specific task and presents a consistent interface to the layer above and below. It is assumed the reader is familiar with the network, transport, and physical layers of this model [6].

RFC 1014 defines the External Data Representation (XDR) protocol that was developed by Sun to present a consistent encoding of data, and byte ordering. Computer systems use different conventions regarding the byte-ordering of data, as well as the format of structures. XDR ensures that one form, called the canonical form is used, which defines one byte ordering, and one structure. The transmission of data using XDR follows the format: sender places local data into canonical format, receiver decodes canonical format to local data.

RFC 1057 defines Remote Procedure Call (RPC). RPC provides a mech-

anism for a host to make a procedure call locally that is actually executed remotely. Typically, a call is executed remotely because the sending host does not have the resources locally to satisfy the call. RPC operates in a client/server relationship where a host with a service is the server, and the host requesting the service is the client. An RPC is packaged in the canonical from using XDR.

RPC is protocol independent and can be used with either TCP or UDP. UDP is usually the preferred choice, because RPCs are short-lived, and match well with the connectionless nature of UDP. The stateless nature of NFS, and the connectionless nature of RPC means that every RPC maintains enough state for that one RPC. This means that every RPC is independent of another and that every operation is self-contained. According to RFC 1057 it is the responsibility of the server to implement execute-at-most-once semantics, to avoid repetitive RPCs causing adverse effects. NFS requests can and are spread across multiple RPCs but the loss of one RPC does not affect the overall stream. A host can retransmit the missing RPC and continue the overall request where it left off.

An implication of RPC being connectionless is that RPC is responsible for implementing its own recovery mechanism in the event of a packet being dropped. Additionally, the arrival of packets out of order must be dealt with properly by some higher level mechanism that understands the semantics of the RPCs connected with a particular service. For example, NFS generated RPCs carry full block number reference state information in each RPC. Hence the server can be stateless.

3.2 NFS

NFS is a stateless protocol; being a stateless protocol the complexity of a server implementation is reduced, but performance is too. Complexity is reduced by the fact every RPC operation carries enough state information for that one request, and in the event of failure only that RPC's effect is lost. For example, if a client is accessing a server, and the server unexpectedly crashes, the client is guaranteed all previously acknowledged RPC operations completed successfully. Performance is degraded by the very reason for increased data integrity, and reduced complexity - statelessness. Having state-packaged and idempotent RPC packets causes additional overhead which always means reduced performance.

There is another performance drawback due to the stateless nature of NFS. All NFS operations must be synced ("synced" is a term used to denoted the writing of all buffers and outstanding data to disk) to disk before they can be acknowledged if data integrity in the event of a system crash is to be preserved. NFS v2 and NFS v3 deal differently with the issue of whether or not a write can be committed synchronously or asynchronously - more on this later. A limited number of RPCs are allowed to interact with an NFS server at a time, and a RPC slot is consumed until the write is committed to disk. Stalling the RPC slot to finish a write makes for much more secure data, but does not help the NFS performance.

NFS clients do not have to be stateless, and most if not all do use state information to increase the performance of NFS. The client caches data and file attributes for quicker access. Client caching does affect the integrity of data because if the host crashes, data in the cache will be lost. Writes cached but not flushed are like local writes in this sense. Cached reads obviously have no impact on integrity.

NFS clients and servers employ specialized daemons called biods (block I/O daemon) and nfsds. A biod is used to pool write operations together into a single request to be sent to the NFS server. This avoids the needless sending of small RPCs, and leads to an overall improvement in performance. A biod connects to a server nfsd with a blocking behavior presented by both.

Typically, the number of biods used by a client is four. Sun recommends[1] for a server with 10 Mbps connection 16 nfsds, a 100 Mbps connections 160 nfsds, and a 1 Gbps connection 1,600 biods. Other tuning guides[2] suggest 16 to 384 NFS threads per CPU, with the actual number being dependent upon processor performance. A SPARCstation 5 should use 16, and a 450 MHz UltraSPARC-II processor should use 384.

In Linux, a biod or rpciod is different than that found in other operating systems. Only a single rpciod kernel thread[3] is used on the client to move data between the client and server. The rpciod is non-blocking as opposed to biods, and is as efficient because it satisfies a request (RPC) up to the point where it can sleep because of pending I/O and goes on to satisfy the next request in the queue. Copying the data from the socket is handled directly by the bottom halves of the networking layer.

[1] *Solaris 8 NFS Performance and Tuning Guide for Sun Hardware (February 2000).*

[2] *System Performance Tuning*, Gian-Paolo D. Musumeci et al, pg 264

[3] As reported by Trond Myklebust, a Linux NFS developer, on the Linux NFS mailing list, http://www.geocrawler.com/lists/3/SourceForge/789/0/8388227/

When a host is not configured to run any biods or nfsds, an RPC connection can still be made by NFS. But, the performance degradation will be large. An RPC connection is typically made from a biod directly to an nfsd, and then later passed into the kernel from the nfsd. If an nfsd is not there to terminate the connection, it is terminated by the kernel, but all further connections will be blocked until completion. Having multiple nfsds to terminate connections avoids this constant blocking, and allows for better performance. If the server is extremely busy, and all of the nfsds become occupied with RPC connections, blocking will occur.

3.3 NFS Versions

NFS is available in three different versions, v2, v3, and v4 [2]. NFS v2 was put into the public domain by Sun Microsystems, and was the first version used in production environments. Subsequent versions of NFS are under the control of the IETF. NFS v3 was developed to address the shortcomings of v2, namely poor write performance, and small block size. NFS v4 is a radical departure from the first two versions - having moved from a stateless to a stateful implementation.

The fact that NFS v2 does not allow for asynchronous writes means the client has to block until a write operation has been successfully committed to disk, and acknowledged. NFS v3 allows for asynchronous writes, which increases the performance dramatically, but it also reduces the data integrity should the server fail.

The block size was increased in NFS v3, which allows RPC connections to bundle much more data, and be more efficient in their transfer size. NFS v2 allowed a block size of 1k or 2k, but v3 added 4k, 8k, 16k, and 32k.

NFS v4 is much different from the previous two versions. It was developed to address the shortcomings observed in NFS as a whole. Specifically, NFS v4 focuses on performance, security, consistency across different computer systems, better access across the Internet, internalization, and localization[4].

[4] *Managing NFS and NIS*, Hal Stern et al., pp 144-146

3.4 Default Testing Conditions

The experiments to be shown are executed with default NFS options. The default options are 8KB block sizes, the UDP protocol, asynchronous write access, and NFS protocol v3. Unless explicitly stated to the contrary these options were the defaults options used for NFS testing.

Chapter 4

NFS Benchmarks

Measuring the performance of NFS in a vacuum is meaningless; there is an obvious need to compare one machine against another, and one way to accomplish this is with industry standard benchmarks. There are two broad categories of benchmarks - benchmarks that closely simulate real-world conditions, and benchmarks that exercise a specific function. The first approach can lead to conditions where faults in the system go undetected because they are obscured by averaging of performance values for the overall system. The second approach allows one to concentrate narrowly on conditions causing performance problems within the system, but may lead to ignorance of other facets of the system or problems only excited by conditions arising from complex semantics arising in actual applications.

The initial benchmark chosen for this project was the industry standard benchmark Connectathon, available from **http://www.connnectathon.org**. The Connectathon benchmark is made up of three separate groups of tests including basic, general, and lock tests. The basic tests consist of simple file operations including file creation, file deletion, directory creation, directory removal, getting attributes, setting attributes, looking up attributes, reading files, writing files, linking, renaming, and getting file system status. The general test executes application programs such as nroff, make and compile. The lock test exercises the ability of the NFS implementation to properly implement locking semantics.

4.1 Connectathon

The first step towards benchmarking the various systems was to establish a baseline. To do this we used different machine configurations utilizing Gigabit Ethernet connections. (Information on the machines used for these tests in Appendix A.) The Connectathon suite was run with different mount options. The mount options used include using no options (baseline uses UDP, NFS v3, and 8KB write and read blocks), specifying NFS v3 and UDP as the protocol, and NFS v2 and UDP protocol. (The default values for baseline were stated both explicitly and implicitly.) The legend of the graphed data is presented with baseline, 3U, and 2U, and should be read as baseline, NFS version 3 with UDP, and NFS v2 with UDP respectively. The reason that TCP was not used as a protocol in these tests is due to the fact that Linux's NFS implementation does not reliably (if at all) support TCP at the time of this writing. The kernel version at the time of this writing is 2.4.18.

The plots used in this thesis connect test points by lines, but in actuality the lines contain no information that is of value to us. However, the readability of the graphs are increased by using lines to connect test results associated with a given test configuration.

The graph shown below in Fig. 4.1 is the output of a Connectathon basic test between Polaris+ (client) and Hutt (server). The basic suite of Connecathon consists of ten different tests. These tests are enumerated in order in the following list.

1. recursive file and directory creation: The number of files, directories, and levels recursed is user specified. The default values are files=5, directories=2, and levels=5. These options can be adjusted at the time of test execution with command-line options.

2. recursive file and directory removal: The number of files, directories, and levels recursed is user specified. The default values are files=5, directories=2, and levels=5. These options can be adjusted at the time of test execution with command-line options.

3. lookups across a mount point: The system getcwd() is executed a user specified amount of times. The default is 250, and can be adjusted at the time of test execution with a command-line option.

4. getattr, setattr, and lookup: Ten files are created in a directory. The system calls chmod(), stat() are executed a user specified amount of

times on these files. The default is 50 for each file and can be adjusted at the time of test execution with a command-line option.

5. write file: A file of user defined size is written ten times. The default size of the file is 1MB. The size of the file is adjustable at the time of test execution with a command-line option.

6. read file: A file of user defined size is read ten times. The default size of the file is 1MB. The size of the file is adjustable at the time of test execution with command-line options.

7. read directory: 200 files are created in a directory and the directory is read 200 times. The directory is operated on with the system calls opendir(), rewinddir(), readdir(), and closedir(). The number of files created, and the number of operations on a directory is adjustable at the time of test execution with command-line options.

8. rename and link: Ten files are created in a directory. Each file is operated on ten times with the system calls link(), and rename(). The number of files created and the number of operation per file is adjustable at the time of test execution with command-line options.

9. symlink and readlink: Ten files are created in a directory. Each file is operated on twenty times with the system calls symlink(), and readlink(). The number of files created and the number of operation per file is adjustable at the time of test execution with command-line options.

10. getfs: The system calls statvfs(), and statfs() are called a default 1,500 times. This value is adjustable at the time of test execution with a command-line option.

The only tests on the graph that can be directly translated into throughput are the write and read tests. These tests were run with the default options. This means Test5 wrote 10MB of data in 0.513 seconds, which translates into 163.5 Mbps. The read test, Test6 read 10MB of data in 0.1 seconds which translates into 838.9 Mbps. Comparing these results to the results gathered with Netperf of 990 Mbps, NFS write throughput behaves more than six times worse.

Read performance throughput was large due to the use of an NFS cache on the client side. Note that the first operation committed was a write, and

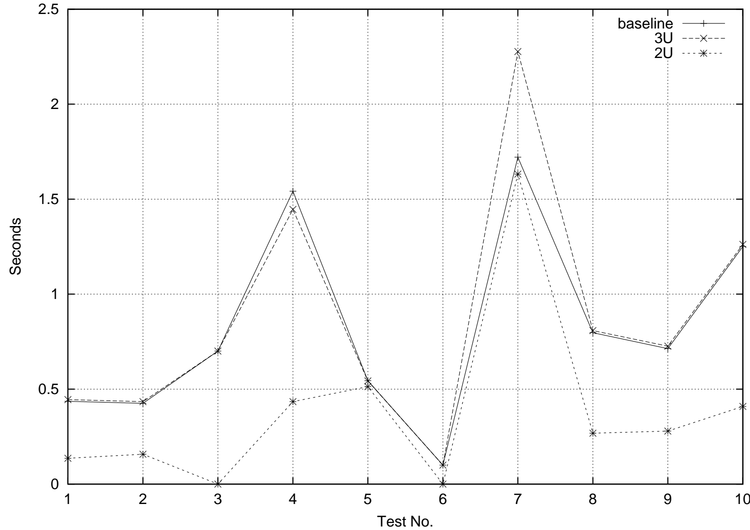

Figure 4.1: Polaris+ to Hutt at 1 Gbps.

the NFS client implements caching thereby retaining the data for later use. The following test is a read test which does not even have to ask the NFS server for the data because it has a copy in its cache thereby producing a large throughput. Read performance out-paces write performance in almost all cases, and results gathered from read tests are skewed from this caching effect. For this reason, read performance is not considered in the examination of NFS we will conduct because its performance is far more reliant on client caching mechanisms.

The large gap in performance that exists between raw Gigabit Ethernet, and NFS write performance led us to believe that the required instruction processing overhead for NFS is large. This large amount of processing should cause throughput to be adversely and linearly affected by reduction of the clients processing power. To test this assertion the configuration was changed to involve the use of a slower client to see if the processing power of a machine did affect performance. Additionally, the bandwidth of the network was lowered to a tenth of its former value to determine if network bandwidth affects performance. These alternative configuration results can be in Fig. 4.2. (These variations were conducted together as a gross determination of their effects on performance. Later in this thesis we will vary the processor

speed separately.)

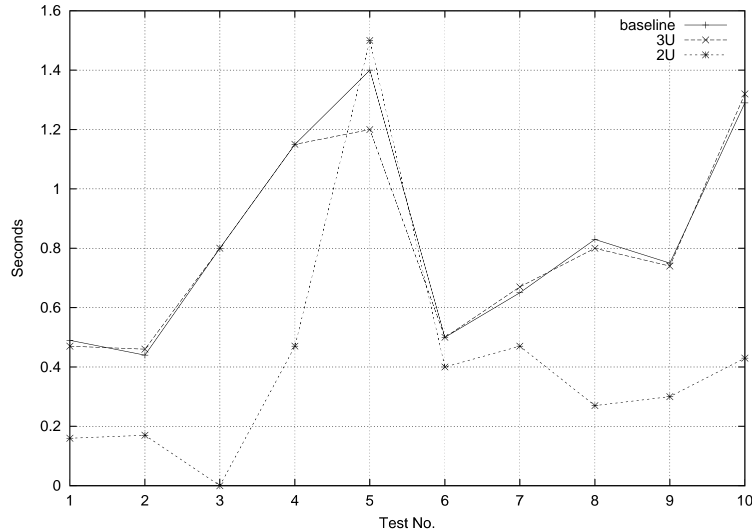

Figure 4.2: Legacy to Hutt at 100 Mbps.

As a result of lowering the processor core clock speed by approximately a factor of four, and decreasing the network's bandwidth to a tenth of its former value, a relatively small effect was observed. The performance of the write throughput experienced a decrease in bandwidth by a factor of 2.3 times, from 163.5 Mbps to 69.9 Mbps.

We expected the performance to decrease by a much larger factor than it did. Our hypothesis that NFS requires more processing power than Polaris+ was able to provide is not valid. That is, our hypothesis that the network was under utilized due to the inadequate processing power of Polaris+ is not valid. The real cause of the observed poor performance might be attributed to a number of possible suspects: biod limitation, nfsd limitation, the transactional nature of NFS/RPC/UDP/IP deadlocks, RPC stop and wait behavior, and the performance bottlenecks having to do with the NIC, and NIC driver.

The results above led us to reassess the nature of the next set of experiments needed to narrow the focus on the possible bottlenecks mentioned above.

The Connectathon suite has many different tests but we choose to concen-

trate on the write throughput values. Tests concerned with get/set attributes were ignored because these types of operations can be accelerated with meta-journaling [11]. The read test was ignored because it is cache implementation sensitive and strongly client dependent. The caching algorithms used, and the amount of memory used to cache is different from client to client. A large cache would allow for more operations to be readily available whereas another implementation with a smaller amount of cache would suffer in comparison.

Our focus is now concerned with write test performance for asynchronous operations so that we can capture the lowest transactional level bottlenecks. Asynchronous mode is used because synchronous mode writes block until the data is committed to disk. This makes tests much more dependent upon the available number of biods, nfsds, and disk speeds. Synchronous writes also introduce a stop-wait behavior because after an RPC for a write is made, the user must wait until the data is committed before continuing. Asynchronous mode allows operations to be outstanding, avoids the stop-wait behavior, and exposes the machine's ability to processes protocol handling from NFS down to the link level as well as expose network related bottlenecks.

4.2 Multiple Tasks

The test results up to this point have involved minimal testing of NFS, but have shown that there is a definite performance issue to be addressed. It has also been shown that the performance of the cache has an effect on the performance of NFS. Before continuing further and testing the various aspects of NFS, and attempting to determine bottlenecks, we would like to show that the caching mechanisms of NFS can be removed. To do this, tests were constructed that eliminated spatial and temporal locality.

To remove spatial locality, large files (32MB) were used instead of the normal 1MB files that Connecathon uses. The use of large files and a large number of tasks should remove spatial locality because the cache cannot maintain enough data before it is forced to commits its contents. Constantly filling the cache's buffer, and forcing the writes to disk makes the principle of spatial locality ineffective. For the same reasons, NFS's cache cannot use temporal locality because it would require too much memory.

Temporal locality can be affected by using many tasks to write many small files. The constant movement of data causes NFS's cache to continuously exchange the contents of its cache with the newer incoming data.

This constant change does not allow NFS to make use of temporal locality, because old data is not retained.

The following tests involve the use of multiple tasks from a single client accessing a single server. The tests were constructed such that a varying number of tasks were executed in parallel, with each one running an instance of the Connectathon write test, Test5. The number of tasks used took on six values for a single test, with the individual break down being 1, 2, 4, 8, 10, 20.

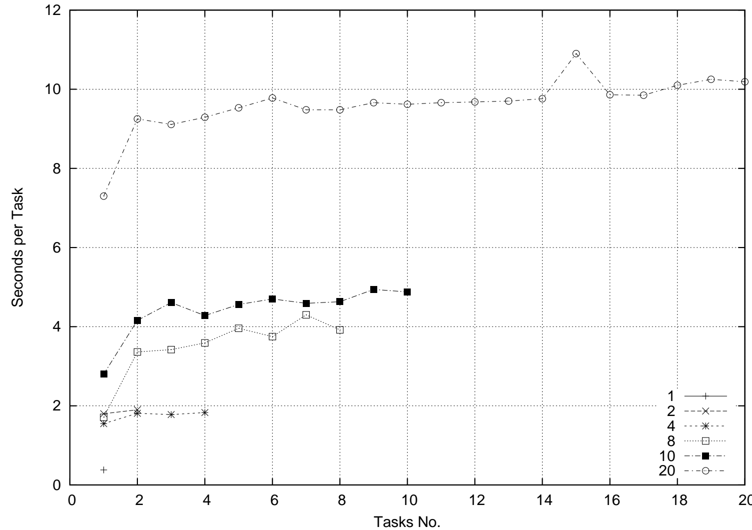

Figure 4.3: Polaris+ to Hutt at 1 Gbps with multiple tasks, and 1MB file size

Fig. 4.3 shows the results of a multiple task test between Polaris+ and Hutt. The graph should be read by reading the X-axis as the index of a task that executes one instance of the Connectathon write test. The Y-axis corresponds to the amount of time required for each task. The results from tasks used in a single test are all tied together with a line to help the reader to easily differentiate between the various tests. Note, when the number of tasks used in a particular test is eight or larger, there is a clear advantage to being the first task. Subsequent tasks require approximately the same amount of time to complete as opposed to the first. This results from a slight advantage had by the first thread having been launched without delay

from a busy processor unlike those that follow. Table 4.1 presents the average throughput for each different task count.

Client	Server	No. of Tasks	Throughput (Mbps)	Standard Deviation (Mbps)
1.7 GHz	700 MHz	1	220.753	0
1.7 GHz	700 MHz	2	120.70	0.007
1.7 GHz	700 MHz	4	48.141	0.13
1.7 GHz	700 MHz	8	23.959	0.79
1.7 GHz	700 MHz	10	19.000	0.61
1.7 GHz	700 MHz	20	8.717	0.68

Table 4.1: Average multitasked throughput between Polaris+ and Hutt with 1MB Files.

These results have a near linear relationship in execution time that can be seen in Fig. 4.4. The data curve is a plot of the actual data. The solid line represents linear performance, and is used to compare the data against linearity.

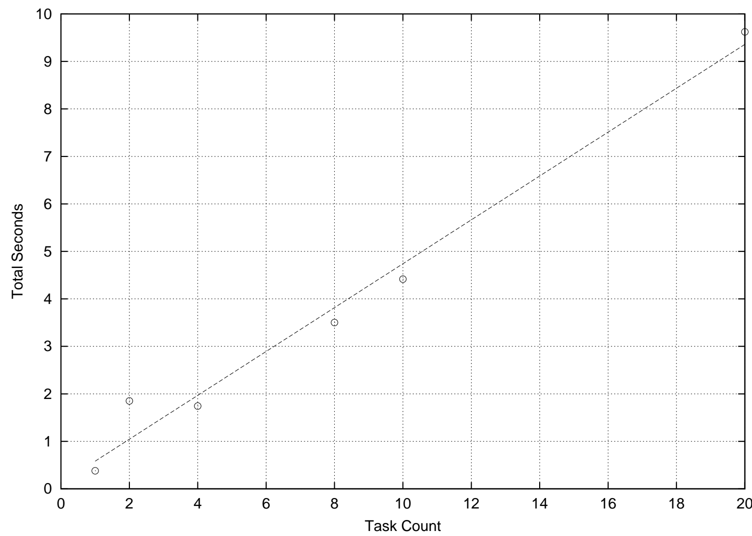

Figure 4.4: Linear relationship as the number of tasks is increased

The Connectathon write test writes a 1MB file ten times in a row. Due to this, subsequent writes after the first benefit from caching. By adjusting the test such that ten different files were written as opposed to one, the benefit of the caching mechanism is obstructed. Fig. 4.5 shows the results of conducting a test between Polaris+ and Hutt using different files for each of the ten file access operations with a size of 1MB, and Fig. 4.6 shows the same test with 32MB files. Table 4.2 and Table 4.3 shows the results with the average throughput based on the number of tasks used for 1MB and 32MB files respectively.

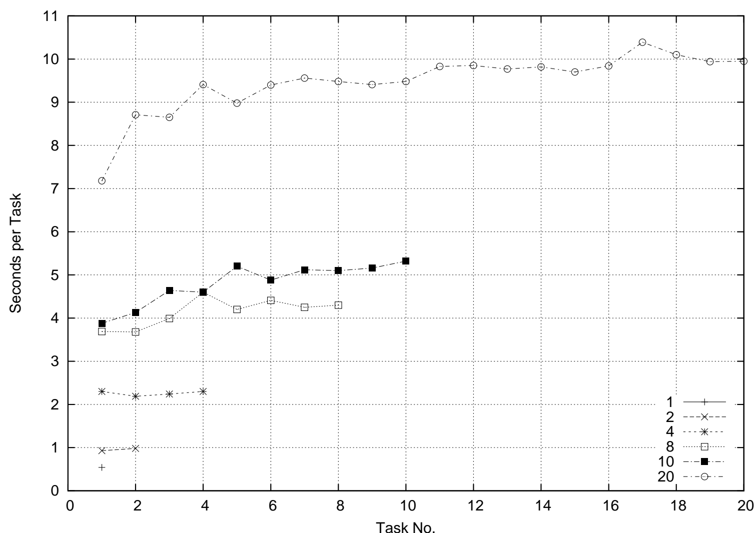

Figure 4.5: Polaris+ to Hutt at 1 Gbps with multiple tasks, and 1MB file size, and different files per each write sub-test.

The results seen in Table 4.3 show that the caching mechanism of NFS is directly affecting the results. Using the same file for all write operations results in a throughput of 220 Mbps, while using separate files the throughput is 153 Mbps; this makes the cached transfer approximately 1.5 times faster than the non-cached transfer. When using 20 tasks the improvement was not seen, but this can easily attributed to the large file size, and the resulting inability to make use of spatial and temporal locality.

The memory required to store 20 tasks each with 32MB worth of data is 671MB - just to handle incoming data for NFS. If this number is adjusted to

Client	Server	No. of Tasks	Throughput (Mbps)	Standard Deviation (Mbps)
1.7 GHz	700 MHz	1	153.345	0.00
1.7 GHz	700 MHz	2	175.68	0.04
1.7 GHz	700 MHz	4	148.64	0.05
1.7 GHz	700 MHz	8	162.10	0.33
1.7 GHz	700 MHz	10	174.65	0.49
1.7 GHz	700 MHz	20	177.11	0.69

Table 4.2: Polaris+ to Hutt at 1 Gbps with multiple tasks, and 1MB file size, and different files per each write sub-test.

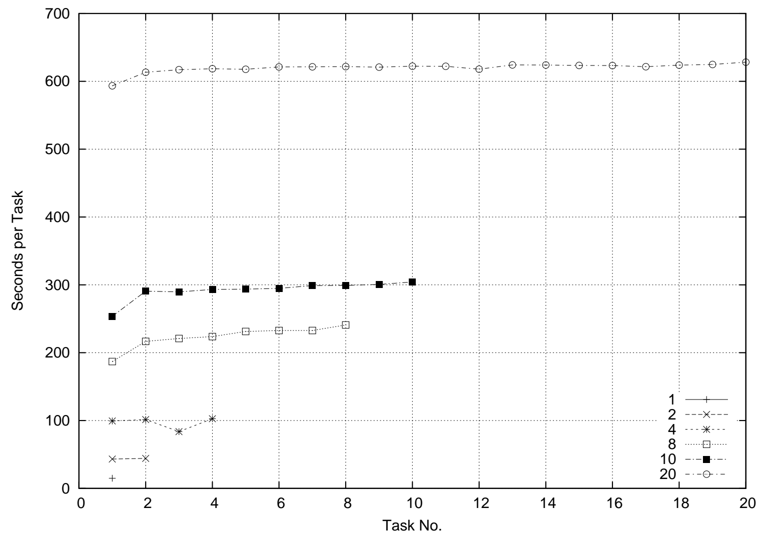

Figure 4.6: Polaris+ to Hutt at 1 Gbps with multiple tasks, and 32MB file size.

Client	Server	No. of Tasks	Throughput (Mbps)	Standard Deviation (Mbps)
1.7 GHz	700 MHz	1	180.521	0
1.7 GHz	700 MHz	2	197.34	0.72
1.7 GHz	700 MHz	4	195.87	8.83
1.7 GHz	700 MHz	8	178.58	16.56
1.7 GHz	700 MHz	10	177.22	14.21
1.7 GHz	700 MHz	20	173.16	7.08

Table 4.3: Polaris+ to Hutt at 1 Gbps with multiple tasks, and 32MB file size.

reflect the writing of ten separate files for each task this number quickly grows to 6,710MB. A cache benefits from frequently accessed areas, and accesses that are temporally close, not long term large requests such as these. Hence from these tests, we have learned that we should use tests in the following experiments that either transfer large files or many different files to truly assess communication speeds without caching effects.

4.3 Loopback

Up to this point all of the tests were conducted over a switched Ethernet network, with 100 Mbps and 1 Gbps line speeds. One of the bottlenecks that may be hampering the performance of NFS is the NIC, and NIC driver performance. The easiest way to eliminate this variable is to configure a host to run in loopback.

In loopback, the host acts as both the client and server, and mounts its own exported filesystem. When using loopback a packet still travels through the TCP/IP stack, but the data-link and physical layer are removed[1]. When a packet reaches the network layer it is placed in the loopback driver, essentially an IP input queue. The IP input queue is a temporary holding place used until the transport layer retrieves the packet. Once the packet is retrieved it travels up through the stack to the application layer.

The tests with multiple tasks were run again with a file size of 1MB and 32MB. The results can be seen Fig. 4.7 and Fig. 4.8 respectively. The results

[1] *TCP/IP Illustrated Volume I*, Stevens, W. R., pp 28-29

are also presented in total throughput for each different set of task in Table 4.4, and Table 4.5.

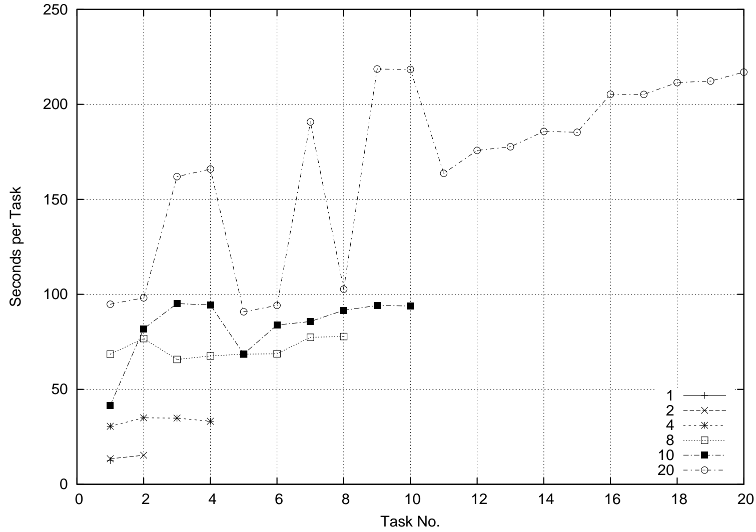

Figure 4.7: Polaris+ Loopback with multiple tasks, and 1MB file size

Host	No. of Tasks	Throughput (Mbps)	Standard Deviation (Mbps)
1.7 GHz	1	524.29	0
1.7 GHz	2	524.29	0
1.7 GHz	4	621.38	0.26
1.7 GHz	8	1,206	0.56
1.7 GHz	10	3,830	0.21
1.7 GHz	20	760.70	0.88

Table 4.4: Polaris+ Loopback with multiple tasks, and 1MB file size.

The results from the loopback tests strongly implicate the NIC and driver as causing the major bottlenecks in NFS, as it can be seen NFS is able to achieve higher throughput when not hampered by the network. The throughput achieved by using 1MB files is almost 4 Gbps at its peak. This number is highly sensitive to the cache, and the average throughput with 32MB files is a more realistic metric. The average of 32MB files was approximately 300

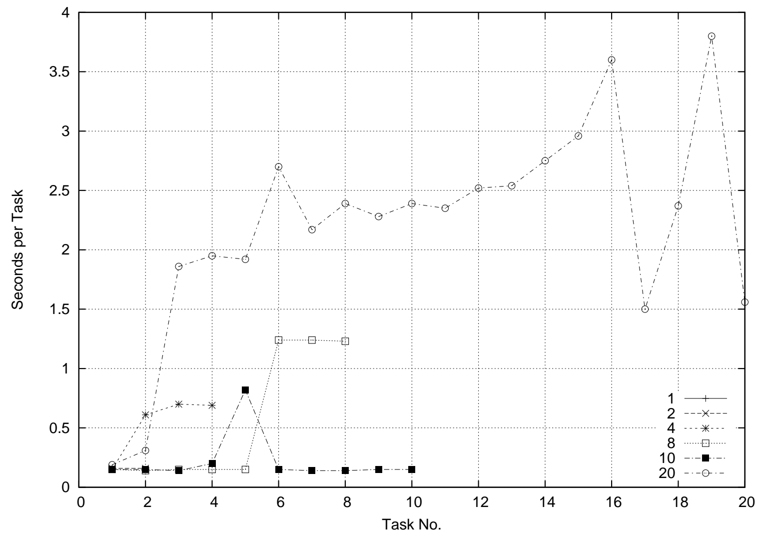

Figure 4.8: Polaris+ Loopback with multiple tasks, and 1MB file size

Host	No. of Tasks	Total Throughput (Mbps)	Standard Deviation (Mbps)
1.7 GHz	1	244.03	0
1.7 GHz	2	374.26	1.31
1.7 GHz	4	321.52	2.05
1.7 GHz	8	300.95	5.01
1.7 GHz	10	323.20	16.76
1.7 GHz	20	318.00	46.67

Table 4.5: Polaris+ Loopback with multiple tasks, and 32MB file size.

Mbps. Recall that the host must do twice the work in loopback therefore; the expected performance of NFS in a host-to-host configuration is greater than 600 Mbps if no execution costs or bandwidth restrictions were associated with the NIC and NIC driver This is clearly higher then our current performance in host-to-host peformance. In fact, the maximum throughput achieved thus far was between Polaris+ and Hutt with a measurement of 197 Mbps (Table 4.3) - more than three times slower.

4.4 Network Latency

Another possible effect on the performance of NFS is network latency. All client server configurations tested go through a single switch (Table 2.1) when creating a network connections[2]. As a result of this minimal distance with respect to the number of intermediary nodes, it is expected that not all NFS servers have the luxury of clients being one hop away, and this section determines if network latency does in fact play a role in performance.

The network configuration for the test can be seen in Fig. 4.9. The client, Polaris+ in this case is connected to the NFS server Hutt, via a single network switch. That switch together with the switch directly below it serves all computers within the ECE department at WPI. The switches to either side of the dashed line are used to interconnect buildings via a 2 Gbps link. This topology is reflective of all tests conducted thus far. The test as seen in Fig. 4.9 wass repeated again, but the topology was changed to that as seen in Fig. 4.10.

The topology Fig. 4.9 yields a ping time of 0.2 ms. The alternative topology yielded a time of 0.6 ms.

After Polaris+ was moved such that it now had to traverse multiple switches to reach Hutt, the basic tests from Connectathon were executed again. The results can be see if Fig. 4.11.

Fig. 4.1 represents the same client and server for the standard topology. These results when compared to those of Fig. 4.11 show that network latency does not have an affect on the performance of NFS. The alternative topology produced write times that were slightly faster the standard topology. They went from 0.544 seconds to 0.518 seconds as a result of switching to a different topology, with a higher latency. This translates into 154 Mbps, and 162 Mbps respectively.

[2]The tests conducted thus far have been on a switched network.

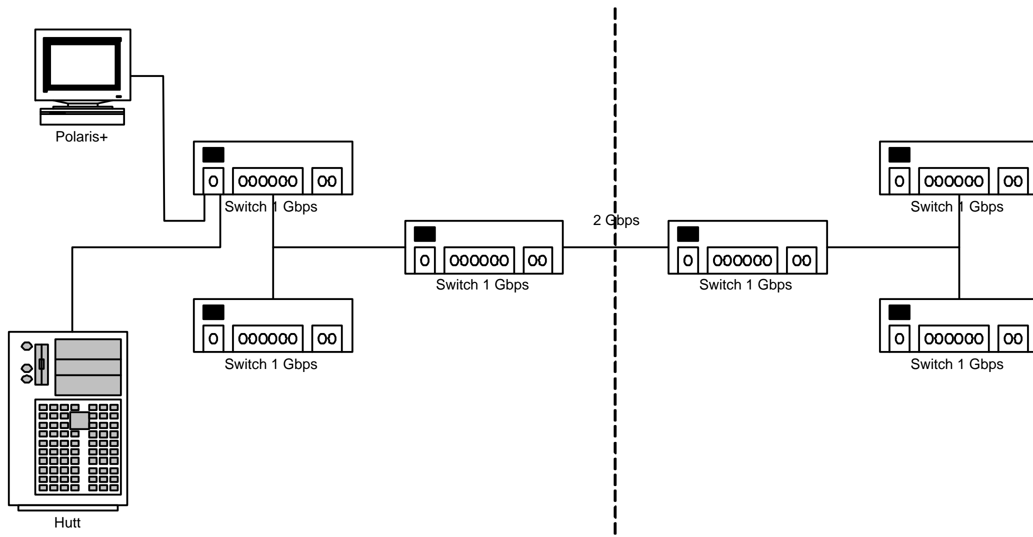

Figure 4.9: Topology used for NFS network testing.

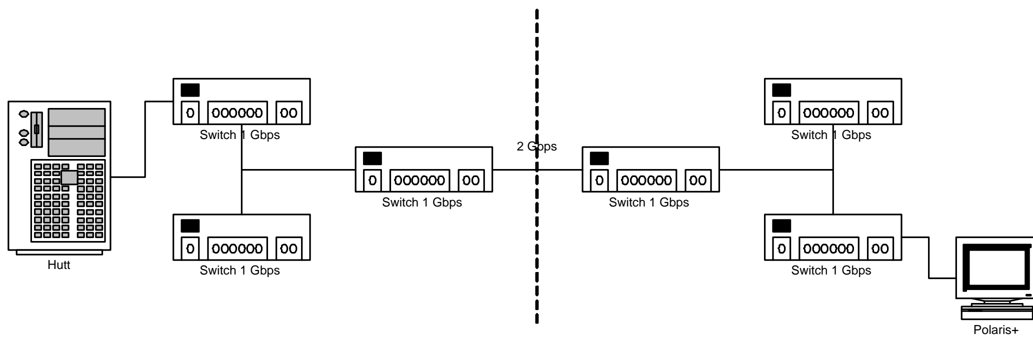

Figure 4.10: Alternative topology used to test network latency's effect.

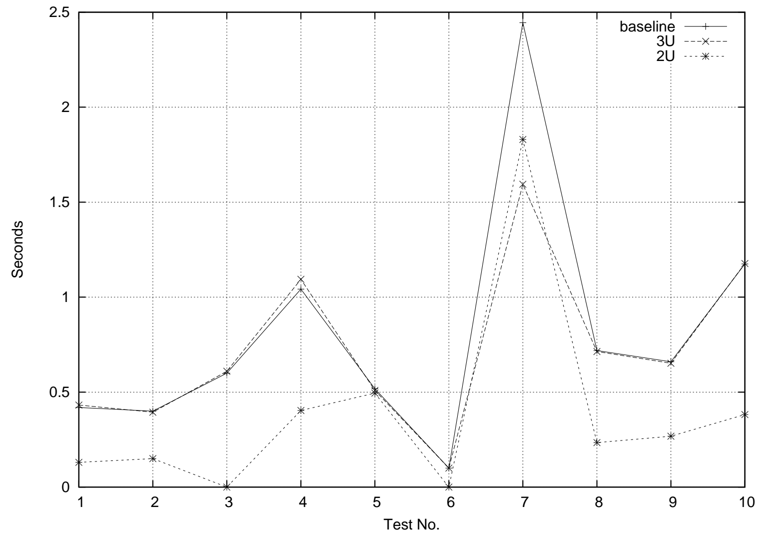

Figure 4.11: Polaris+ to Hutt at 1 Gbps through alternate topology.

As will become evident later, these results were misleading and caused us to subsequently not identify the true basis of our performance bottleneck until some contradictory results arose later.

Chapter 5

An Interim Performance Model

5.1 Processor Bound Performance Model

A phenomenological mathematical model will now be developed from the data gathered to this point in the investigation to gain better insight into the protocol processing and packet propagation mechanisms at work in an NFS connection. The model is also used to develop a better understanding of the performance parameters of each host under test. The model accurately describes the fixed processing time of a packet, and the processing time that is dependent upon the size of the overall packet. Logically, fixed processing can be thought of as the amount of time a packet must always incur when moving through the TCP/IP stack. The variable processing time would be due to things such as calculating a packets checksum, which is shorter for smaller packets and longer for larger packets.

The model does not describe solely the amount of time it takes to process a single packet, but rather the time to process a packet, plus an associated gap between packets. The NIC places a packet on the wire. Once that occurs the NIC is able to move onto the next packet to be serviced, but before it is able to launch the next packet a small amount of time has elapsed. This elapsed time creates a small gap between transmitted packets. Fig. 5.1 illustrates this decomposition.

The loopback test demonstrated what seems to be a bottleneck associated with the action of sending a packet across the network. In loopback mode NFS was able to achieve a throughput of 524 Mbps (Fig. 4.7), while an NFS connection to another machine yielded only 221 Mbps (Table 4.1). Since

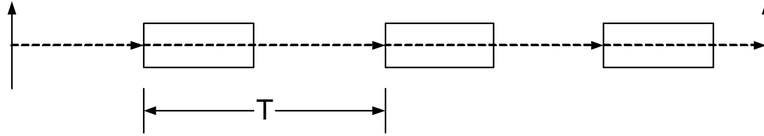

Figure 5.1: Packet inter-arrival times compared to packet processing times.

earlier tests had shown that the NIC driver and card could sustain throughput of rates of greater than 990 Mbps, the location of the bottleneck would appear to be in the processing required to prepare the packet for launching since the loopback interface used an MTU of 16,384 bytes and the Ethernet connection used an MTU of 1,500 bytes. A likely suspect is the processing time required to fragment the 9,000 byte NFS write requests into packets that fit the specified MTU. These results suggest that performance of the NFS system is processor performance bound. If this were the case, variation of the processor load required to launch packets should have a direct and easily modeled effect on throughput. The fact that the MTU of an IP interface can be changed at will providing a simple means to vary the load associated with each NFS transaction.

The purpose of this test is to determine the processing power of the client's TCP/IP stack, how long it takes to process a packet based on its length, and the fixed overhead experienced by every packet.

5.2 MTU Variation

The results were gathered by connecting several different hosts over NFS. We used both a network connection via Gigabit Ethernet, and a loopback connection. The Connectathon write test with 1MB files, and one task was then executed on the client with successive iterations adjusting the MTU. By using a small file size we seek to suppress disk transfer response effects and highlight protocol processing. The MTU was then adjusted to three distinct values. The first set of data used to verify our model was collected from Polaris+ in loopback mode. The overall throughput of the test, and MTU were recorded. Table 5.1 shows these results.

Using the arguments presented above, suppose the overall processing time is dependent upon the MTU size, then a fixed processing time exists. A

MTU	NFS Throughput (Mbps)	R = MTU/s	T = 1/R (μs)
576	363	78776	12.6
1500	444	37000	27
16384	600	4577.6	218

Table 5.1: Connectathon write test on Polaris+ through loopback.

variable amount time also exists that changes as the size of the packet does. T_0 represents this fixed amount of time, and α represents the proportionality constant that yields a variable amount of time. M represents the size of the MTU is bytes.

$$T(M) = \alpha M + T_0 \tag{5.1}$$

We can obtain a testable (though not optimal) model by using two of three values from Table 5.1 T_0 and α can be solved.

$$\alpha 16384 + T_0 = 218$$
$$\alpha 1500 + T_0 = 27$$
$$\Rightarrow \alpha = 0.0128325, T_0 = 7.75 \mu sec$$

After the two unknowns have been calculated verify the results with the first results - an MTU size of 576.

$$\text{Test: } \alpha 576 + 7.75 = 15.14 \mu sec$$

To further verify the model, the test was run again with Polaris+ acting as an NFS client, and Hutt acting as an NFS server. The collected data for throughput for a given MTU is presented in Table 5.2.

MTU	NFS Throughput (Mbps)	R = MTU/Sec	T = 1/R (μsec)
256	128.6	64,300	15.5
576	156.86	34,040	29.3
1000	166.67	20,825	48

Table 5.2: Connectathon write test, Polaris+ to Hutt at 1 Gbps.

Solving the values in Table 5.2 reveals the following:

$$\Rightarrow \alpha = 0.0441, T_0 = 3.9\mu sec$$

These values are on the order of those found with Polaris+ in loopback for the results in Table 5.1. This T_0 is approximately half of what was found for loopback (7.75), which is as one would expect because a machine in loopback has to do twice the amount of work.

The outcome of this test seems to verify the conjecture that performance is processor bound and related to a large cost which is proportional to the number of MTUs processed. Furthermore, the MTU proportional cost is significantly increased by the process of passing datagrams from the IP stack to the transmitted NIC hardware and back again at the receiver.

Further tests were conducted with matched and higher speed clients and servers to confirm this hypothesis. We ultimately found this model to not be correct via these tests.

Chapter 6

An Interim Performance Model Refutation

6.1 Disk Speed

This section presents Connecathon tests across which disk performance was verified. The purpose of these tests is to determine the role that disk speed has in the processing of NFS requests. Multiple tasks writing 1MB and 32MB files were used to conduct these tests. The same file was written ten times for all tests, and each task. The speed of each disk was determined by using hdparm[1], a tool used for the configuration of IDE disks, which also has facilities for raw disk access performance. Three different disk speeds were used including 2.89 MB/sec, 11 MB/sec, and 42 MB/sec. The two slowest speeds represent the same disk with the slower of the two having DMA transfers turned off. This was done using hdparm.

The values stated above for disk speeds refer to time it takes to transfer data from the disk without benefit of caching mechanisms as measured with hdparm.

The first test was against the slowest disk with a transfer speed of 2.89 MB/sec. The result for this test with 1MB files, and 32MB files can be found in Fig. 6.1 and Fig. 6.2 respectively. The maximum throughput achieved for the test shown in Fig. 6.1 is 264 Mbps. This number clearly shows that disk performance had no effect on the performance of NFS due to caching mechanisms.

[1] http://www.linux-ide.org/hdparm-fdisk.html

The result of the test shown in Fig. 6.2 clearly show that when caching mechanisms are removed the performance of the disk becomes an upper bound on NFS performance. The throughput achieved by 20 tasks was approximately 23.0 Mbps.

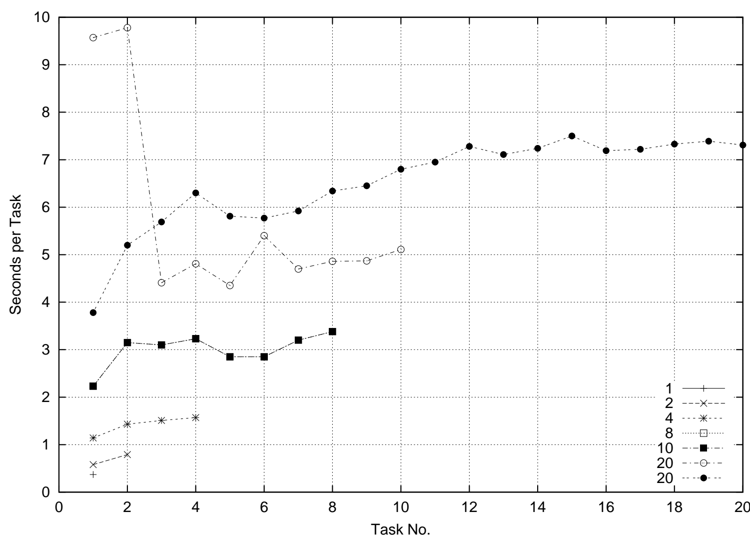

Figure 6.1: Revanche to Beast at 1Gbps, multiple tasks, 1MB file size, and disk with 2.89 MB/sec transfer speed.

The same tests were repeated again for a disk speed of 11 MB/sec. The results for this test with 1MB files, and 32MB files can be found in Fig. 6.3 and Fig. 6.4 respectively. The maximum throughput achieved for the test shown in Fig. 6.3 was 243 Mbps. This result should be comparable to those found in Fig. 6.1 because disk performance was shown not to be a factor in NFS throughput. As a result of increasing the file size to 32MB the throughput for 20 tasks was 62.8 Mbps, or approximately 2.7 times the performance found the disk with a speed of 2.89 MB/sec. The difference between the 2.89 MB/sec disk, and this disk is 3.8 times.

The last set of results for the measurement of disk performance with a disk speed of 42 MB/sec can be seen in Fig. 6.5 for 1MB files and Fig. 6.6 for 32MB files. The maximum throughput achieved in Fig. 6.5 was 227 Mbps. This result matches well with the previous results for 1MB files, and multiple tasks by indicating NFS cache mechanisms are the main factor in

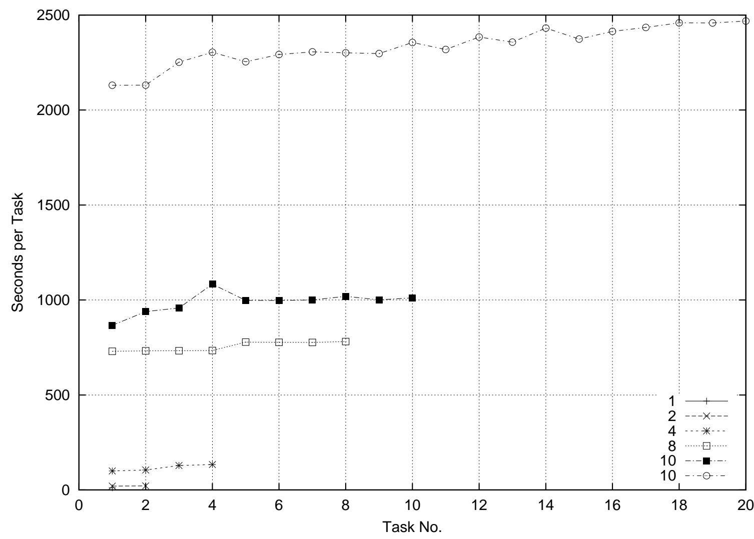

Figure 6.2: Revanche to Beast at 1Gbps, multiple tasks, 32MB file size, very slow disk.

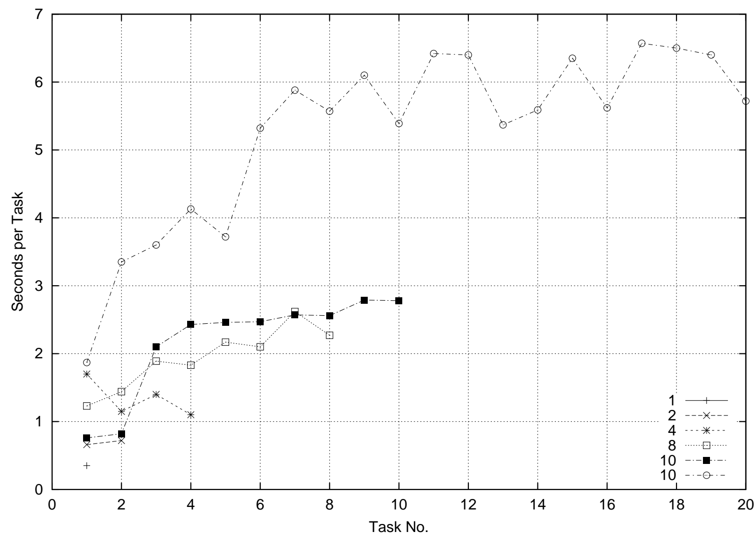

Figure 6.3: Revanche to Bastion at 1Gbps, multiple tasks, 1MB file size, slow disk.

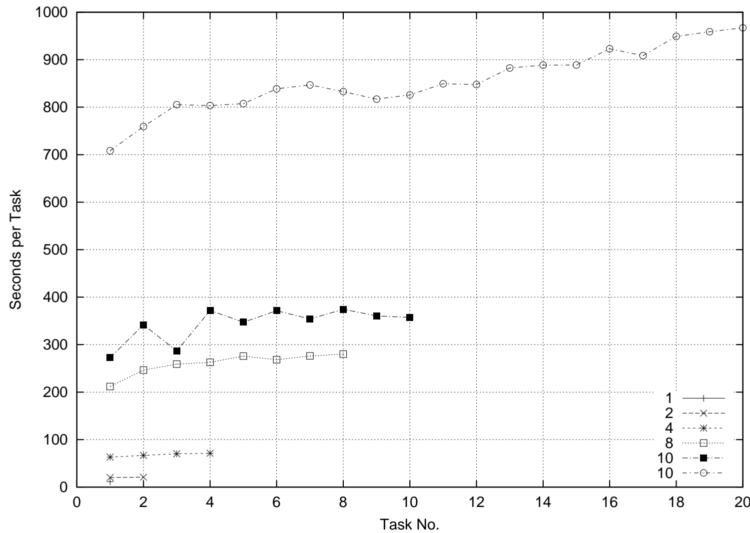

Figure 6.4: Revanche to Bastion at 1Gbps, multiple tasks, 32MB file size, slow disk.

the performance for small files. The increase in file size to 32MB, and using 20 tasks resulted in a throughput of 115 Mbps.

The results of these tests give better insight into NFS's caching mechanisms. The results show that the performance difference between all of the disks is actually quite small when using 1MB files, even 1MB files with 20 concurrent tasks. The use of a slow disk greatly effects performance as load was increased. This is as one would expect because the cache is able to hide the fact a slow device is actually being used to service requests.

Table 6.1 presents a summary of test results for 32 MB files for all three disk speeds.

The first point at which a considerable divergence in performance across the three disks is noticed upon using 32MB files, and four tasks. The slowest disk has an average test completion time of approximately 117 seconds. This value is nearly 2.5 times worse than the fastest disk, and 1.5 times worse then the disk with a performance of 11 MB/sec. Either the cache is still having an effect on performance, or their exists another bottleneck not yet considered. That is because, if only disk speed was taken into account the ratio between the slowest disk and fastest disk should be 14.5 times.

The next point to consider is made by the results the case with the largest

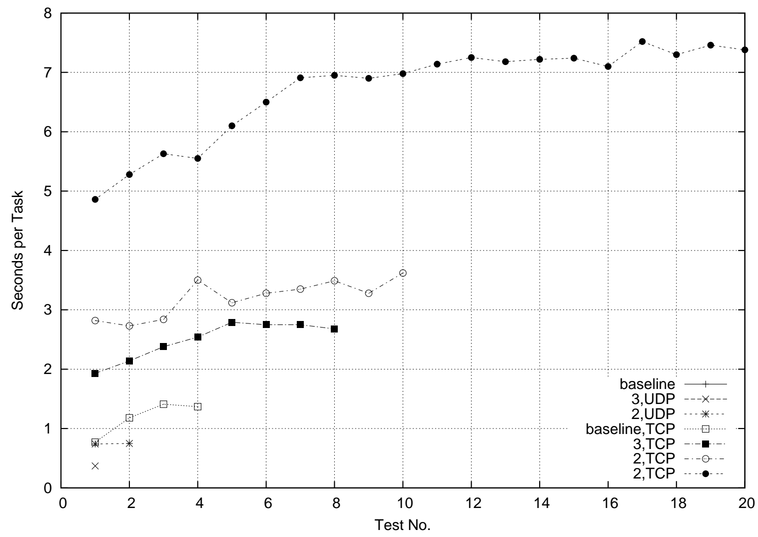

Figure 6.5: Revanche to Beast at 1Gbps, multiple tasks, 1MB file size, fast disk.

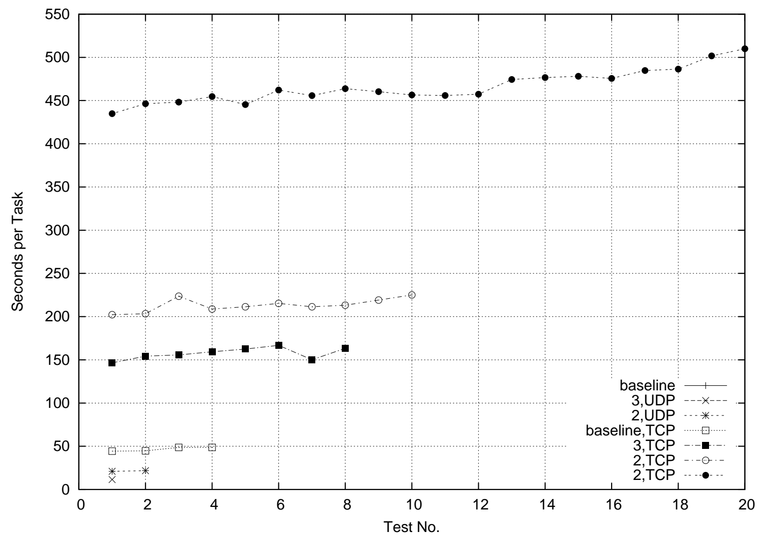

Figure 6.6: Revanche to Beast at 1Gbps, multiple tasks, 32MB file size, fast disk.

Disk Speed	Server	Tasks	Avg. Time	Throughput (Mbps)	Standard Deviation (Mbps)
2.89	Beast	1	11.28	237.97	0
11	Bastion	1	11.50	233.42	0
42	Beast	1	11.45	234.44	0
2.89	Beast	2	20.69	259.55	1.05
11	Bastion	2	20.59	260.81	0.46
42	Beast	2	21.38	251.11	0.59
2.89	Beast	4	116.93	91.83	16.65
11	Bastion	4	67.85	158.26	3.66
42	Beast	4	46.62	230.32	2.41
2.89	Beast	8	755.27	28.43	24.66
11	Bastion	8	260.04	82.58	22.37
42	Beast	8	157.33	136.50	6.93
2.89	Beast	10	987.35	27.19	57.00
11	Bastion	10	343.80	78.08	35.54
42	Beast	10	213.38	125.80	7.70
2.89	Beast	20	2,336	22.98	97.93
11	Bastion	20	855.45	62.76	66.50
42	Beast	20	466.43	115.10	19.26

Table 6.1: Throughput with 32MB files, and multiple tasks for different disk speeds.

number of tasks and largest file size. The slowest disk results in test completion services requests on average is 2,336 seconds. The fastest disk is able to complete the test in 466 seconds. The results for the fastest disk are that the tests execute approximately 5 times faster than for the slowest disk. Considering the large amount of data that must be moved, it is difficult to believe caching as any effect on such a large number of tasks, and data. Therefore we must consider this to be strong evidence that yet another mechanism is at work.

6.2 FreeBSD

All of the tests conducted thus far have been done exclusively with Linux clients and servers. This opens the possibility that performance is solely the effect of the underlying operating system, and poor NFS performance is due to Linux, and not another factor. To prove to ourselves that NFS's poor performance is not limited to Linux a FreeBSD client and server were used. Information for these machines can be found in Appendix A.

Kaboom (client, 1 GHz), and Crash (server, 266 MHz) were connected over a 100 Mbps link via a Netgear FS105 switch. The Connecathon basic suite was used to execute the test. The results for this test can be see in Fig. 6.7. The throughput achieved was 88.7 Mbps.

The results of this test are similar to the results obtained in Fig. 4.2, which paired a Linux client and server together over a 100 Mbps line. FreeBSD executed the Connectathon write test in a time of 0.951 seconds, whereas Linux took 1.4 seconds. The throughput achieved was 70.5 Mbps. We account for the better performance of Crash (266 MHz) and Kaboom (1 GHz) versus Legacy (400 MHz) and Hutt (700 MHz) to the processing power of Kaboom as this test is cache dominant.

The performance of FreeBSD running on both the client and server confirmed that NFS bottlenecks exists when using other operating systems. The performance of NFS when run between different operating systems was also considered to learn if any anomalies exists between different NFS implementations and operating systems. The results of this test can be seen in Fig. 6.8.

The results from Fig. 6.8 show the performance of writes to be between those of using a FreeBSD client and server, and a Linux client and server. FreeBSD hosts had a write time of 0.951 seconds, Linux hosts had a time of

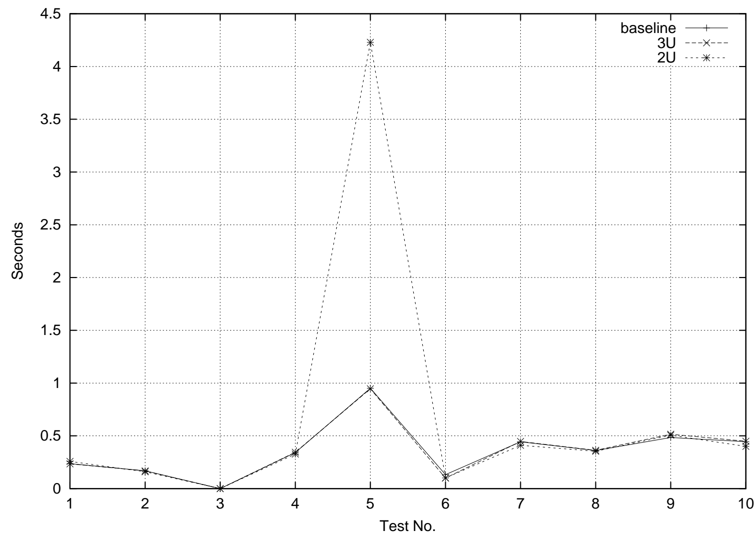

Figure 6.7: FreeBSD: Kaboom to Crash at 100 Mbps.

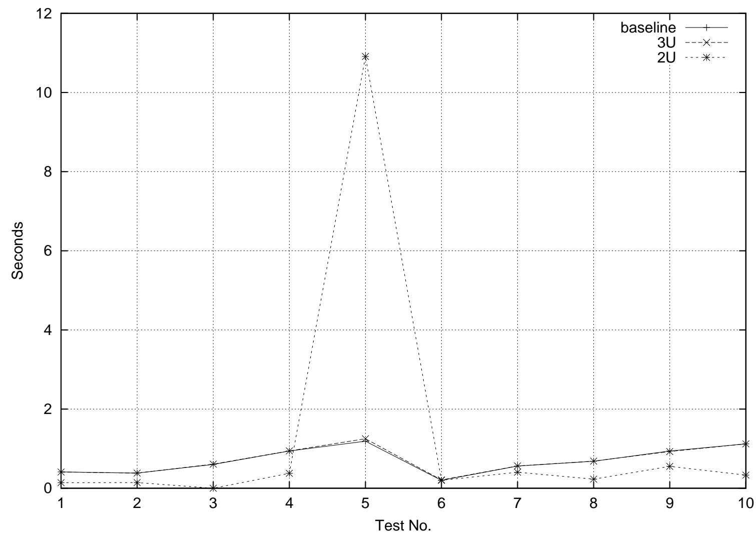

Figure 6.8: (Linux) Kaboom to Crash at 100 Mbps.

1.4 seconds, and a Linux client, and FreeBSD server had a time of 1.1 seconds, or 76.3 Mbps. Fig. 6.7 and Fig. 6.8 show that NFS performance bottlenecks cannot be solely attributed to Linux due to relatively equal performance seen from different operating systems, and intermingled clients and servers.

The last test sequence exposes the fact that NFS throughput as measured in these tests is a strong function of both client and server processing performance, led to our construction of the machines Beast and Revanche. Beast and Revanche, as can be seen in Appendix A are identical in every hardware and software aspect.

By applying tests to such a matched pair we can specify test results with respect to a single stated processor speed, FSB speed, etc. Furthermore, it could allow us to use symmetry arguments when processing delays need to be assigned to one host or the other for certain situations, as for example those exemplified by the Interim Performance model of Chapter 5.1

With the capabilities of the new hardware discussed above and lessons learned about the effects of disk performance and caching we revisited and improved several important measurements. The following chapters will discuss these new results and use them to develop a model for NFS transactions which finally sheds light on the performance problem.

Chapter 7

Round Trip Time Benchmarks

Chapter 4 served to verify that mechanisms exist that hinder NFS performance. Chapter 4 also dispelled possible bottlenecks such as the operating system, and the inability to avoid NFS caching skewing collected results. Looking outside of the NFS, the next logical step is to exam the underlying network, and network components. It needs to be determined if the network or NIC plays a role in affecting overall NFS performance. It is necessary to understand how the operating system, TCP/IP implementation, NIC, and network contribute to the overall performance of NFS. To do this a simple test suite was constructed to measure the round trip times (RTT) for both UDP, and RPC as well as the file transfer time using UDP.

7.1 UDP and RPC RTT

The first constructed test suite consisted of two UDP programs. The first UDP program measures the RTT between hosts by measuring the amount of time it takes a client to send a packet to the server, and the server to respond. The second UDP test involve an ftp-like transfer between a client and a server. The client sends an amount of packets needed to total 1MB and 32MB worth of data. Once the server has received all of the data it responds with a sentinel packet, when the sentinel packet is received the time is recorded.

The reason these two different tests are needed is that the first test measures the time between packets being launched from the NIC (as discussed in Section 5.1), a measure of the computational overhead of the operation,

whereas the latter tests the maximum throughput possible.

One of the key components of these simple benchmarks are their ability to precisely measure packet times. A typical approach would involve the use of the UNIX system call *gettimeofday*. The resolution of gettimeofday is different from platform to platform, and can vary from as high as 1 to as low as 10,000 microseconds. The Pentium platforms have a 64-bit register that increments every clock tick called the Time Stamp Counter (TSC). This counter can be read using the rdtsc assembly instruction. Polaris+ has a calculated clock speed of 1.7 GHz allowing for a timer resolution of approximately 0.6 ns. The source code for these benchmarks can be found in Appendix B.

Packet Size	Time (μs)
576	320.13492
1000	353.50824
1500	382.20932

Table 7.1: UDP round trip times between Polaris+ and Hutt.

Table 7.1 shows UDP RTT between Polaris+ and Hutt when connected via a Gigabit Ethernet switch versus packet size as varied by setting different MTU values for the interface being used.

Packet Size	Time (μs)
576	343.792
1000	376.250
1500	448.917

Table 7.2: RPC round trip times between Polaris+ and Hutt.

Table 7.2 shows RPC RTT between Polaris+ and Hutt.

Table 7.3, shows the results of monitoring the transmission of 1MB and 32MB worth of 1,500 byte packets to a server with the server transmitting a sentinel packet at the end of transmission. The total time was then recorded upon completion. The amount of time it takes to transmit 32MB versus 1MB is approximately 31 times greater, indicating a near linear relationship as one would expect.

Size	Seconds	Throughput (Mbps)
1MB	0.014433136	581.20
32MB	0.447929	599.28

Table 7.3: UDP throughput with FTP-like transfers.

The results from the UDP and RPC round trip times exhibited a near linear behavior with respect to packet size. This behavior can be seen in Fig. 7.1, and Fig. 7.2. The overhead associated with RPC is small compared to that of UDP.

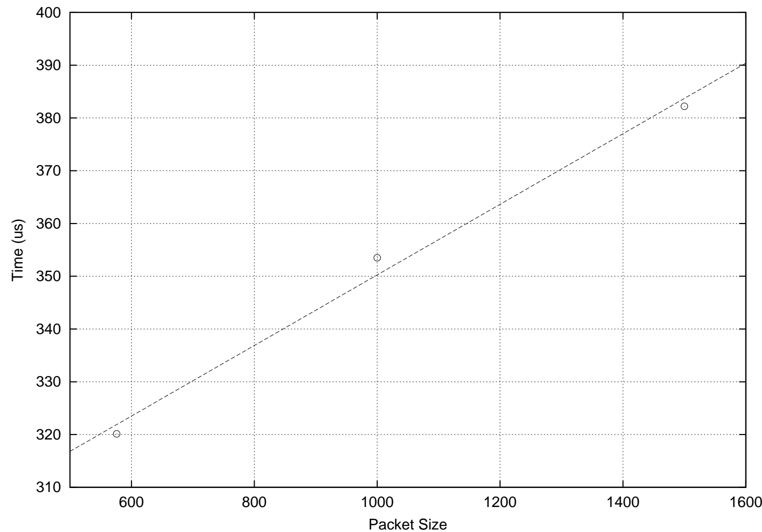

Figure 7.1: Near-linear relationship between packet size and UDP RTT.

These results confirm the general linear relationship between core processor clock speed and throughput of our interim model. However, the above results clearly indicate that our previous conclusions in Chapter 5 concerning the large increase in processor overhead associated with pushing datagrams through the NIC and driver are unfounded. The throughput achieved with 32MB of data is almost 600 Mbps as opposed to 221 Mbps for the same machine when conducting an NFS based transfer as seen in the results found in Table 4.1.

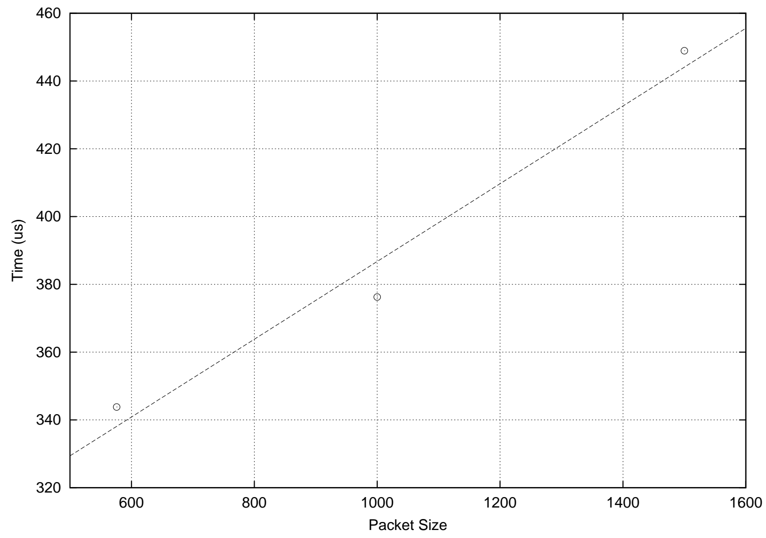

Figure 7.2: Near-linear relationship between packet size and RPC RTT.

Chapter 8

IP Fragments

One possible cause for the large discrepancy between loopback and host-to-host performance is lost IP fragments. The loss of an IP fragment requires a complete retransmission of the original datagram. This potentially creates a cycle whereby fragments are constantly being lost causing more datagrams to be retransmitted. The cycle of fragment loss, and retransmission causes poor network performance.

8.1 IP Fragmentation

RFC 1191 defines path MTU (PMTU) as the maximum packet size a network is able to transmit or receive. If the packet size transmitted is larger than the MTU then it is the responsibility of the network layer to fragment the packet. A fragmented packet is not reassembled until reaching the final destination, but it is possible for an IP datagram to be fragmented more than once. IP fragments are reassembled at the network layer of the receiving host. The fragmentation and reassembly at the network layer makes it transparent to the transport and other higher layers, such as TCP/UDP.

The IP header has different fields to indicate the fragmentation status of a datagram. The *identification* field contains a unique number for each transmitted packet. This field is copied into each fragment for a particular datagram. All fragmented packets have the *flag* “more fragments” set except for the last packet of a fragmented datagram. The *fragment offset* (in 8-byte units) indicates the offset of this packet in relation to the original datagram. The *total length* is changed for each datagram to represent the fragmented

packets size (RFC 791).

A host can specify that a datagram should not be fragmented by setting the "don't fragment" bit in the *flags* field. If this flag is set and the datagram has to be fragmented it is instead dropped, and an ICMP error is sent to the originating host.

With datagram fragmentation, independent packets are created. Each packet has its own IP header, and is routed independently of other packets. This makes it possible for packets to arrive out of order. The end host should allocate enough buffer space for out of order packets for reassembly of the entire datagram.

Even though fragmentation is transparent to upper layers there is a pernicious problem associated with ostensibly transparent operation. If an IP fragment is lost during transmission the entire datagram must be retransmitted. This happens because IP does not have retransmission or timeout, these are functions of the higher layers. Another reason is that the source host cannot determine the particular fragment that it needs to have retransmitted. If a fragmented packet was fragmented in transit, the receiving host would have no knowledge of this. TCP does implement retransmission and timeout, but UDP does not. In the case of NFS over UDP, RPC recovers lost IP fragments by retransmitting entire RPC requests upon timeout.

The timeout period is defined at mount time, in tenths of a second with the option *timeo*. For NFS over UDP, this parameter is a minor timeout, meaning RPC uses exponential binary backoff by doubling *timeo* until the number of retransmission threshold is reached. The retransmission threshold value is also specified at mount time[1].

The RPC retransmission algorithm is not able to distinguish between a congested network and a slow host, hence it is not able to deal intelligently with these situations. If an RPC is retransmitted it is irrelevant if the RPC was lost or still enqueued in the NFS server[2]. It is the responsibility of the server to enforce execute-at-most-once semantics if retransmission causes duplicate RPCs (RFC 1057).

[1] *Managing NFS and NIS*, Hal Stern et. al, pg 430.
[2] *Managing NFS and NIS*, Hal Stern et. al, pg 429.

8.2 Linux IP Fragments

Linux provides user controlled variables to alter the behavior of the operating system. These variables can control the virtual memory (VM), networking, drivers, and running processes. The documentation for these variables can be found with the standard distribution of the Linux kernel under the Documentation directory (see ./Documentation/sysctl for more details).

Handling of IP fragments is controlled by three variables, *ipfrag_high_thresh*, *ipfreg_low_thresh*, and *ipfrag_time*, the documentation for these variable can be found in Linux's Documentation directory (./Documentation/networking/ip-sysctl.txt). *ipfrag_high_thresh* controls the maximum amount of memory allocated for the reassembly of fragmented packets. When *ipfrag_high_thresh* is reached fragments are discarded until *ipfrag_low_thresh* is reached. *ipfrag_time* is the amount of time in seconds to keep IP fragments in memory. The defaults for these variables can be found in Appendix C.

8.3 Variable Adjustment

The performance of NFS can be increased by utilizing larger block sizes for RPC requests[3]. However, the larger the block size the more IP fragments that are generated as a result. (A block size of 8KB is typical.) As blocks are transmitted at high rates of speed, the probability of IP fragment loss increases. These losses can be the result of the high transmission speeds, and the inability of the OS to buffer all incoming packets, and IP fragments for reassembly. As a result of a packet being lost RPC must request the entire packet sequence, or datagram again. (RPC must make the request if the underlying protocol does not retransmit. Recall, UDP is the popular choice when using NFS, and does not have retransmit based reliability.) Additionally, these retransmissions can fail just as the original datagrams did, and have their IP fragments lost. The constant cycle of retransmission and IP fragment loss can greatly degrade the performance of NFS.

The number of received IP fragments is recorded by Linux in its proc filesystem[4]. The number of fragments that could not be reassembled is also recorded.

[3] *System Performance Tuning*, Gian-Paolo D. Musumeci et. al, pg 260.

[4] Located in /proc/net/snmp

An apparent way to boost performance, and remove the need for fragmentation is to use a larger MTU size such that fragmentation is avoided. An MTU of 9,000 is able to completely encapsulate the 8KB block size of an NFS transaction block size and obviate the need for fragmentation. Increasing the buffer space for IP fragment reassembly should also reduce the number of retransmissions, unless the option of increased MTU size is not available.

8.4 Testing for IP Fragments

In loopback mode, it is expected that the host has little to no packet loss. Transmitting packets over a network yields a much greater potential for packet loss. The state of the network could be congested, cause the introduction of large latency, and introduce other adverse effects.

An 8KB NFS block must be fragmented into six parts before it can be transmitted via an ordinary Ethernet LAN connection, and if an IP fragment is lost the entire 8KB block must be retransmitted. Performance is further hampered by the fact that UDP is the protocol for communication, and RPC mechanisms must be relied upon for reliability. The performance of RPC in this regard compared to TCP is poor[5]. TCP also has the ability to dynamically adapt to network conditions, handle congestion, and adjust transfer sizes.

To determine if fragmentation was affecting performance four different tests were executed. The four tests make use of the Connectathon write test, and use two different file sizes 1MB, and 32MB. One set of tests writes the same file ten times, this allows the test to benefit from NFS's cache. The second set of tests writes ten different files, decreasing the benefit of the cache. These tests were executed with a crossover cable between two copper Gigabit Ethernet NICs as described in Table 2.1. The results for these tests can be seen in Table 8.1.

The above test was repeated using switched connectivity over fiber. The results may be seen in Table 8.2.

There is a notable difference between crossover and switched connectivity. Test four, the test that exhibits the least skew due to caching effects, shows a difference of approximately 7% seemingly indicating that either the crossover connection or the copper based adapters provide better performance. The

[5] *Managing NFS and NIS*, Hal Stern et. al, pg 436.

Test Type	ReasmFails	FragCreates	Throughput (Mbps)
Write 1MB, Same	0	7,680	225.50
Write 32MB, Same	0	821,760	381.08
Write 1MB, Different	0	829,440	226.11
Write 32MB, Different	0	1,321,050	137.22

Table 8.1: IP fragmentation results for direct crossover connection.

Test Type	ReasmFails	FragCreates	Throughput (Mbps)
Write 1MB, Same	0	7,680	217.89
Write 32MB, Same	0	821,760	311.84
Write 1MB, Different	0	829,440	213.45
Write 32MB, Different	0	1,321,050	127.90

Table 8.2: IP fragmentation results when connected through a Gigabit Ethernet Switch.

fact that no reassembly errors were seen however seriously brings the current conjecture into doubt.

The results shown in Table 8.1 serve as a baseline to see if an increase in performance is observed as a result of eliminating fragmentation. Fragmentation is eliminated by using 9,000 byte jumbo frames to completely encapsulate the 8KB NFS block. A surprising fact from the data is that no packets were lost as a result of fragmentation.

Even though no fragments were lost, the ability to process a large amount of fragments could still be hampering performance. The same test was repeated again with the use of jumbo frames, and an MTU size of 9,000 bytes. The number of fragmentations, and resassemblies does not appear in the table because the larger frame sizes obviates the need for fragmentation. The result of using jumbo frames on performance can be seen in Table 8.3.

The results shown in Table 8.3 prove IP fragment processing is not a burden. Performance did increase, but only minimally. The first test went from 225.50 Mbps to 253.43 Mbps, a 12% increase in throughput as a result of not processing IP fragments. The last test executed writes 32MB of data into 10 separate files causing over a million fragments in the test without jumbo frames, and yet the performance actually decreased by 1.5% as a result of not processing IP fragments when using jumbo frames. This result

Test Type	Machine	Throughput (Mbps)
Write 1MB, Same	Beast	253.43
Write 32MB, Same	Beast	403.48
Write 1MB, Different	Beast	248.18
Write 32MB, Different	Beast	135.12

Table 8.3: Jumbo frame performance through a direct crossover connection.

indicates that the processing performance of each end host is large enough that the processing of IP fragments does not present a bottleneck in NFS performance. An under-powered host should see a much larger benefit as a result of using jumbo frames, and avoiding fragmentation and reassembly.

8.5 MTU's Effect on Performance

It has been show that the use of jumbo frames does not have a large effect on the performance of NFS. However, we wanted to determine if there existed any anomalies for different MTU sizes that did effect performance.

The following test adjusted the MTU of Revanche for the loopback interface in increments of 128 bytes, up to the default value of 16,384 bytes. For each iteration a single task was executed that wrote 32MB of data into ten different files. The throughput was then recorded. The results of this test can be seen in Fig. 8.1.

The performance of NFS in loopback with a varying MTU is very erratic. There are two places of interest on this graph. The first is for MTU values of less than 1500 where a definite trend of performance with respect to MTU size can be seen. The second, is for MTU values larger than 8,500, where a small increase in performance is seen, this can be attributed to the fact that 8KB NFS blocks no longer need to be fragmented. The overall erratic nature of the graph for higher MTUs can most likely be explained by the effect of the current activity of the machine, when MTU processing is no longer the dominant CPU load.

The same test was repeated between Revanche and Beast with the MTU adjusted in increments of 128 bytes for each host. The results can be seen in Fig. 8.2.

The results of this graph show that NFS performance is slow with an

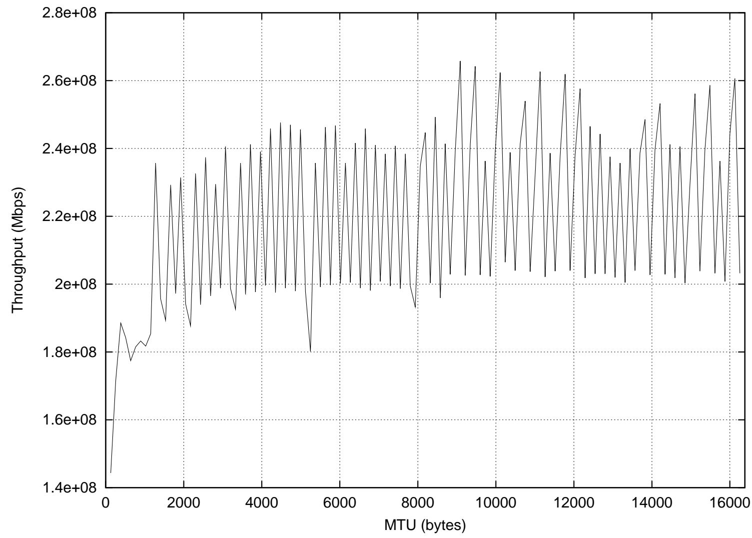

Figure 8.1: Loopback performance of Revanche for varying MTU sizes.

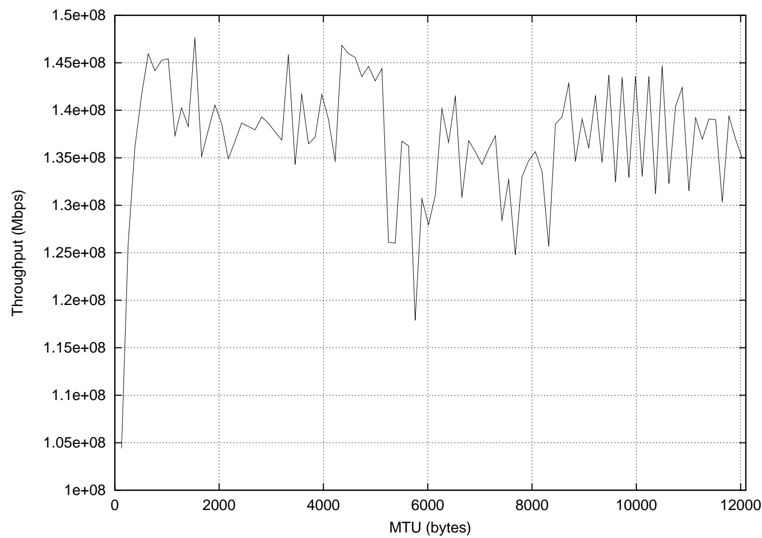

Figure 8.2: Beast to Revanche performance for varying MTU sizes connected directly by a crossover cable.

MTU of less than 1,500 bytes. NFS performance was at its peak for an MTU of 1,536 bytes, and 4,352 bytes. The closest MTU value to the standard size of a jumbo frame (9,000 bytes) with a large throughput was 10,496 bytes which achieved the eighth largest throughput of all MTU values. The results are close to what one would expect. An MTU value of 1,536 is very close to Ethernet's MTU hence only required fragmentation was performed. An MTU of 4,352 is approximately half of the space consumed by an 8KB NFS block that was used for this test.

The peculiarity of this graph is the poor performance of MTU's that range is size from approximately 5,700 to 8,300 bytes. We believe this can be attributed to the block size of NFS. Splitting an NFS block in half yielded approximately 4,000 bytes, but it appears a threshold has been crossed where a large portion of the block is being sent in the first Ethernet frame, and then an increasingly smaller portion in subsequent frames. However, once the entire NFS block is encapsulated in one MTU the performance quickly increases, as can be see starting at an MTU value of approximately 8,300 bytes.

It has been shown that the elimination of IP fragmentation through use of Jumbo frames does not improve the performance of NFS. Current processor speeds are more than adequate to absorb the effect of IP fragmentation and reassembly, but jumbo frames may still be of benefit to low performance clients.

Chapter 9

A Transactional Model

The results, tests, and explanation developed thus far have left a conundrum. According to our tests a host-to-host connection cannot achieve network "wire" throughput because there exists a bottleneck in the system. However, the Netperf benchmark test results indicate that the network hardware and software implementation is more than capable of delivering the desired performance. This leads us to believe that a transactional problem with NFS must exist. But, the loopback test sheds doubt on this because we are able to achieve from 2.5 to 3.5 times the performance of the full network by exercising only a small portion of the transactional route as shown in Section 4.3. In this chapter we introduce another model inspired by our collection of data that reveals the nature of the performance bottleneck.

9.1 Core Processor Speed

The speed of a processor is determined by two things the FSB (Front Side Bus), and the CPU's multiplier. The FSB setting determines the clock speed of the memory system on a motherboard. The multiplier is a fixed value within the CPU that the FSB is multiplied by to determine the clock speed of the processor. For example, a FSB of 90 MHz and a multiplier of 11.5 would result in a processor speed of 1,065 MHz. Usually, the FSB is not controllable by the user, and for good reason. Adjustment of the FSB above rated values can lead to components overheating, and becoming permanently damaged. Lowering the FSB below rated values can lead to synchronization problems for system components. The adjustment of the FSB, when sup-

ported, can be done through a computer's BIOS, or via jumpers on the computer's motherboard.

The performance of NFS has been shown to differ depending upon the processing power of the machine. However, this dependence is stronger for a loopback mode connection than that for a host-to-host connection. To better observe this phenomenon, Beast and Revanche's FSB speed were adjusted. For each modification of the FSB a test was run and the results recorded. The FSB of both machines were changed such that each machine had the same setting. A loopback test was also conducted for each iteration of FSB. The machines used in this test were constructed with the same parts, and are identical. The processor's multiplier for each machine is 11.5.

The tests conducted involved sequentially writing ten different files each with 32MB of data, and varying the MTU over values of 1,500, 9,000, and 16,384. The MTU value of 16,384 was not used for the host-to-host connection because Ethernet's 32-bit CRC is reliable only up to 12,000 bytes, and the NIC's do not support values above 16,000. The test results for both loopback and host-to-host were combined onto one graph and can be seen in Fig. 9.1.

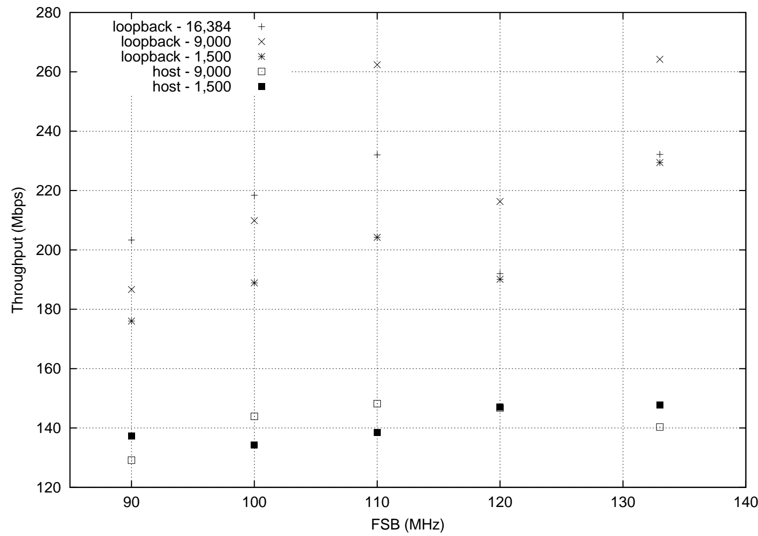

Figure 9.1: Loopback and host-to-host performance for different MTU sizes for Revanche and Beast.

The host-to-host performance appears to have hit a limit, where perfor-

mance does not increase with an increase in processing power. This in contrast to earlier tests in this thesis which were performed on slower machines. The loopback configuration still displays a slight increase in performance with processor speed. Overall, the 50% increase between a FSB of 90 MHz and 133 MHz yielded a performance increase of 14%. Our conclusion from this test is that the bottleneck in network performance is not due to insufficient processor speed.

9.2 Network Latency

When we first took a look at network latency and its effect on NFS performance only the basic suite of Connectathon was used. However, later tests have shown that, to develop a better understanding of performance, a more rigorous treatment is needed, such as using multiple tasks, large files, and writing different files to remove the caching effect of NFS. To remedy the previous lack of attention to this behavior a more developed argument is given.

From Chapter 7 it was determined that the round trip time of a packet with an MTU of 1,500 bytes was 382.21 μs. Approximately half of this value gives us the propagation delay between Polaris+ and Hutt. The test was run again between Revanche and Beast over both a crossover and switched connection. The round trip time over the switched connection was 194.14 μs, and for the crossover connection it was 176.85 μs. The size of the packets was kept as small as possible (46 bytes for Ethernet) to isolate the delay of the network as much as possible.

If a large packet had been used serialization time for the network would have skewed the actual round trip time. Additionally, using RPC instead of UDP would have forced us to account for the semantics involved in an RPC connection, and the latency associated with an RPC connection versus that of a UDP connection.

A total of 10,000 trials were taken, with the average of the trials used as the round trip time. We found that the minimum was within 2% of the average, and that the maximum was larger than the average by approximately 20 μs for both a switched and crossover connection. This implies that the switch is effecting the measured round trip times.

9.3 Disk Latency

The hard drives of today are able to store large amounts of data. The performance of hard disks however has made little headway when compared to that of processor and memory performance.

In Table 6.1 we saw that disk speed can have a significant impact on NFS performance in certain configurations. On the other hand, one interpretation of the data in Fig. 9.1 is that disk speed is not responsible for our primary performance problem since a loopback configuration ought not to enjoy significant performance increase over host-to-host configuration as seen.

Hard disks use LBA (Logical Block Addressing) to access data from the physical disk. An LBA corresponds to a 512 byte block of data. The NFS block size used throughout our testing was 8KB, which is equivalent to 16 blocks of hard disk data. Furthermore, the raw speed as evaluated by hdparm (Section 6.1 of the disk in Beast is equivalent to 42 MB/sec. The time it takes to access 8KB worth of data from the disk is 186 μs based upon this I/O speed estimate.

When the hdparm test reads data from the hard disk it only reads blocks from the disk that are not in the buffer cache. This ensures that the raw speed of the disk is measured, and that it is not influenced by any caching mechanisms.

The raw throughput of the disk as reported by hdparm is not a good metric when used to compare disk performance for the purpose of our experiments. As blocks are written to the disk they are ordered optimally to reduce seek time to write the data to the disk. The algorithm used to order the blocks is called an elevator algorithm[1]. The elevator algorithm is applied to all data committed to the local disk. Likewise, as NFS writes data to the disk the data is ordered optimally by the same elevator algorithm to minimize seek times.

To observe the raw performance of the disk with minimized seek times, a modified Connectathon write test was used against a local disk. The test involved sequentially writing 32MB files, ten times, with ten unique file names. This test was run against a disk with a performance of 42 MB/sec, and 23.11 MB/sec as reported by hdparm, these results are presented in Table 9.1.

[1] *The Design and Implementation of the 4.4 BSD Operating System*, Marshall K. McKusick et al., pg 198

Host	hdparm (MB/sec)	Connectathon (MB/sec)
Revanche	23.11	33.24
Beast	42	46.55

Table 9.1: Comparison of hdparm and Connectathon measured write throughput.

Table 9.1 which reports the disk performance as measured using hdparm, and a Connectathon write test shows the expected throughput increase experienced by the Connectathon test. We are interested in the time it takes to write a block to the hard disk as experienced by Connectathon, hence, we will be using the Connectathon measures in our subsequent analysis. To write 16, 512-byte blocks (this is equivalent in size to a 8KB NFS block) to Revanche's disk takes 246.23 μs, and for Beast's disk it takes 175.98 μs.

9.4 Model of NFS Operations

The latencies for the disk and network as determined previously will now be used to model an exchange between a client and server. Fig. 9.2 illustrates the agents involved in this model. The sequence of events and latencies that describe this model can be seen in Fig. 9.3 and Fig. 9.4.

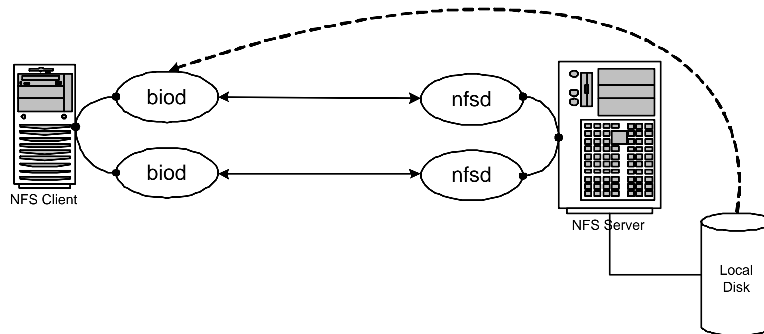

Figure 9.2: NFS model of a transaction.

This model is a simple representation of an NFS client communicating with an NFS server. The client has two biods that match up with the server's

two nfsd. To emphasize disk latency the server hard drive has been depicted as being an external component of the server when in fact it is internal.

The model, as drawn shows the possibility for two RPC transactions to occur concurrently. The large propagation delay of the network allows for an RPC to be processed as another one is being transmitted to the NFS server. The time required for a pipelined request is 264 μs + T_{op}, as opposed to the time for a non-pipelined request 416 μs + T_{op}. These two intervals are differentiated by a dotted line in the diagram.

As will be shown later our collected data does not support the idea that Linux's NFS implementation effectively pipelines RPC requests for increased performance. Though this is contrary to the intended design and operation of NFS, we will later discuss how this behavior arises under heavy traffic loads.

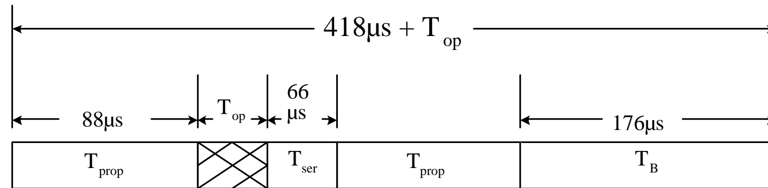

Figure 9.3: NFS latency diagram for sequential transfer with a host-to-host configuration.

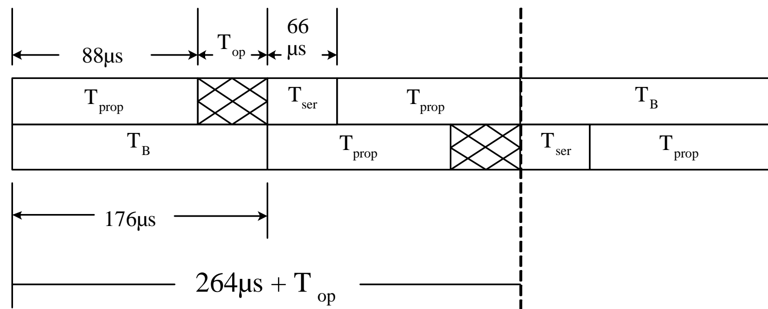

Figure 9.4: NFS latency diagram for two concurrent transfers with a host-to-host configuration.

Fig. 9.3 breaks down the chain of events according to the amount of time

necessary to complete each operation assuming sequential transfers. Fig. 9.4 describes the same actions as Fig. 9.3 except it makes use of two concurrent transfers. The times used in this diagram are for a disk with a raw speed of 42 MB/sec, and the network connection medium is fiber through a Gigabit Ethernet switch.

We will now explain the origin of the parameters T_{prop}, T_B, T_{ser}, and T_{op}. T_{prop} represents time associated with network driver and NIC latency, and the propagation delay of the link. T_B represents the time associated with retrieving data from the hard disk for one NFS 8KB block. T_{ser} is the amount of time needed to place an 8KB NFS block onto the network (serialization time). T_{op} is an unknown quantity that represents the overhead time needed for a host to process each NFS block transfer as well as the processing time associated with RPC, UDP, and IP.

The most immediate observation one should make about this diagram is the amount of time consumed waiting between NFS requests. Assuming a sequential transfer model the latency of the network, and disk dominate the total time of a transfer.

If the host-to-host transaction diagram with a sequential transfer mechanism is correct then there are many other implications of this timing diagram for NFS requests. It clearly explains why better throughput performance is seen between loopback, and host-to-host connections. This is obviously due to the lack of propagation and serialization delay introduced in loopback mode.

The adjustment of the FSB (Fig. 9.1) caused no change of the performance for host-to-host connections, and yet loopback did see a small increase in performance. The introduction of jumbo frames (Table 8.3) produced a small effect on the overall throughput of host-to-host connections, but this can be attributed to the relatively small size of T_{op} when compared to that of the other variables.

For example, in Section 8.4 we measured the impact that jumbo frames had on performance. We saw that an NFS test that was forced to fragment IP packets had a performance of 137.22 Mbps, and the same test repeated with jumbo frames had a performance of 135.12 Mbps. This result further illustrates the small role of processing time compared to that of network and disk latency. The serialization time of a jumbo frame is less than that of a fragmented datagram, because redundant information must be inserted into each packet created as a result of the fragmentation. The performance of the NFS server and client used for the test was large enough that the benefit of

jumbo frames had essentially no effect on the throughput of NFS.

The difference observed in network throughput achieved with Netperf compared to that of NFS can be attributed to the fact that Netperf is not transactional and hence does not experience stop and wait behavior. When Netperf is prosecuting a benchmark, either the client is streaming data, or the server is, and neither host is waiting for the other one before continuing onto the next operation. The fact that Netperf does not need to complete transactions allows it to effectively ignore latency, and continue to pump data onto the network.

The relatively small difference that was observed as a result of decreasing network bandwidth by a tenth, and the processing speed by a third (Section 4.1) can be attributed to the dominance of network latency in this model. The latency did not decrease between hosts, but the value T_{op} did increase, due to the poor performance of the client. However, T_{op} would have to grow large to become the dominant factor in NFS performance.

9.4.1 Loopback NFS Latency Model

The model for loopback is a simplification of the host-to-host model in that latencies associated with the network are removed. Diagrams describing loopback are presented in Fig. 9.5, and Fig. 9.6.

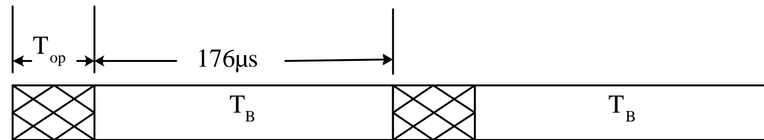

Figure 9.5: NFS latency diagram, loopback mode for sequential transfers.

Fig. 9.5 describes the performance of an NFS loopback mode connection assuming sequential transfers. The host is idle during periods of disk transfer even though it has the ability to process another request.

Fig. 9.6 describes the performance of an NFS loopback mode connection assuming concurrent transfers. With this type of configuration the host is free to process other requests to ensure they are ready to write once a previous disk access has finished.

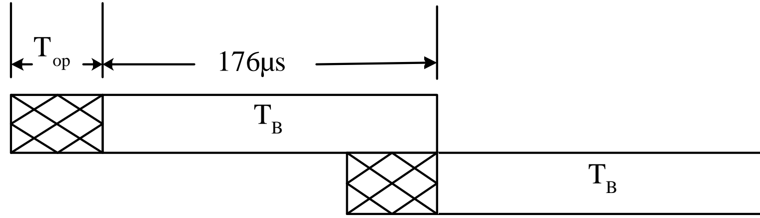

Figure 9.6: NFS latency diagram, loopback mode for concurrent transfers.

9.5 Calculations

The two models for network latency present a compelling case for the unanswered and confusing questions presented. This section codifies our results, and verifies the accuracy of the host-to-host and loopback model with sequential transfers through calculations.

9.5.1 Host-to-Host

Equation 9.1 describes a formula to determine the total delay experienced as a result of a host-to-host NFS connection given the sequential transaction model.

$$T_{cycle} = 2 \cdot T_{prop} + T_{op} + T_{ser} + T_B \tag{9.1}$$

$2 \cdot T_{prop}$ was determined in Section 9.2 by finding the round trip time between Revanche and Beast for both a direct crossover connection and a switched connection. $2 \cdot T_{prop}$ for a crossover connection was 195.20 μs, and $2 \cdot T_{prop}$ for switched connectivity was 176.85 μs. The test used the UDP protocol with a one byte payload in the packet. The measured latency takes into account the processing time of the UDP/IP stack whereas we are only concerned with the latency of the network. Measuring the latency of a UDP connection in loopback mode eliminates the network latency component and allows us to measure the processing time of the UDP/IP stack. The measured latency of the UDP/IP stack was 0.732 μs in loopback mode. Applying the latency of the UDP/IP stack to the calculated latencies for a switched and crossover connection yields 192.68 μs and 175.39 μs for the round trip time respectively.

The formula for calculating T_{ser} can be seen in equation 9.2.

$$T_{ser} = \frac{Network\ Bit\ Rate}{Block\ Size} \tag{9.2}$$

In equation 9.2, the network throughput is 1 Gbps because that is the rated speed of the adapters used. The values calculated using Netperf for the network performance are not used because they take into account UDP/IP processing time. We are concerned with the amount of time to put data on the wire, which for an 8KB NFS block at 1 Gbps is 65.536 μs. This time is represented by T_{ser}. This value actually represents a lower bound on the performance because it only takes into account the payload, and not the control data associated with NFS, RPC, UDP, and IP. Because the payload is dominant compared to control data, the addition of the control data is not considered in T_{ser}.

The parameter T_{op} cannot be determined using the data from a host-to-host connection but it can be determined using measured data in a loopback configuration. The results will later be amended to take into account the calculated T_{op}.

Tests conducted thus far have evidence to support that T_{op} is indeed small. According to our model T_{op} is small compared to delay, and an improvement of T_{op} means little when compared against the other times that make up T_{cycle}. This was perhaps most evident by adjustment of the FSB (Fig. 9.1) and observing the performance of loopback and host-to-host configurations.

Solving equation 9.1, and equation 9.2 yields results that can be seen in Table 9.2. The comparison of our calculated results based on our latency model were made against tests that removed NFS caching mechanisms. The observed and measured values are presented in Table 8.1 for crossover connectivity, and Table 8.2 for switched connectivity. The calculation for these values is illustrated in equation 9.3.

$$T_{cycle} = \frac{8{,}192\text{B NFS Block} \cdot 8\text{bits}}{2 \cdot T_{prop} + T_B + T_{ser} + T_{op}} \tag{9.3}$$

The calculated throughput values are larger than the measured values as would be expected because we have not yet accounted for T_{op}. We have also made the assumption that NFS does not employ concurrent transfers, and

Connection	$2 \cdot T_{prop}$ (μs)	T_{ser} (μs)	T_B (μs)	Calc. Sequential (Mbps)	Calc. Concurrent (Mbps)	Measured (Mbps)
Switched	192.68	65.54	175.98	150.94	240.66	134.48
Crossover	175.39	65.54	175.98	157.19	248.55	137.69

Table 9.2: Host-to-host latency NFS performance calculations between Revanche and Beast based on $T_{op} = 0$ approximation.

support for that assumption is evident in the data presented in Table 9.2. If NFS were using concurrent transfers, we would expect the measured result to be much larger than our calculated results for which the throughput was calculated with the assumption that concurrent transfers do not occur.

The calculated throughput represents an upper bound on the performance of NFS without concurrency. Taking into consideration the latency of the network, disk, and serialization time and ignoring the processing of NFS, an upper bound may be found.

9.5.2 One NFSD

Our data shows that Linux's NFS implementation either does not have support for or ineffectively supports the concurrency of RPC requests. The purpose of this section is to specifically test that hypothesis by removing the ability to form concurrent RPC requests by only making a single nfsd available to the client. The number of nfsds used by NFS can be changed in Linux. In the Debian Linux distribution, the number of nfsds started is controlled by the init script nfs-kernel-server. This will vary from Linux distribution to distribution.

A Connectathon write test was conducted between Beast and Revanche consisting of ten unique 32MB files written sequentially to a server. The average throughput over ten trials was recorded. The test was also run between Revanche and Beast to ensure that results for two different disk latencies were collected. Recall, that Beast has a hdparm measured disk throughput of 42 MB/sec, and Revanche had a throughput of 23.11 MB/sec. The test was repeated in loopback for both Beast and Revanche. Finally, the tests were repeated for switched and crossover connectivity to observe the effect of different propagation delays. These results are presented in Table

9.3.

Connection	Client	Server	hdparm throughput (MB/sec)	Measured Through-put (Mbps)	Connectathon local speed (Mbps)
Switched	Revanche	Beast	42	128.08	372.41
Crossover	Revanche	Beast	42	137.58	372.41
Loopback	Revanche	Revanche	23.11	250.20	270.55
Switched	Beast	Revanche	23.11	119.50	270.55
Crossover	Beast	Revanche	23.11	125.03	270.55
Loopback	Beast	Beast	42	275.35	372.41

Table 9.3: Effect of different network latencies and disk speeds on performance with a single nfsd.

The results found in Table 9.3 can be compared against those found in Table 9.2. Table 9.2 represents a crossover connection between Revanche and Beast, with Beast having eight nfsds running. Comparing the performance of eight nfsds, and a single nfsd for a crossover connection the difference is slight with eight nfsds having an increase in performance of 0.11 Mbps (from 137.69 Mbps to 137.58 Mbps). The performance difference for switched connectivity was slightly greater with eight nfsds having an advantage of 6.4 Mbps over that of one nfsd (134.48 Mbps and 128.08 Mbps).

The results presented in Table 9.3 will be used later in determining the accuracy of our model once T_{op} is calculated in the next section by providing a variety of conditions with which we can test our model.

9.5.3 Loopback

The loopback diagram (Fig. 9.5) presents a simplistic model with a simple solution. There is one unknown, T_{op}. We know T_B, which represents the time needed to read a block of data from this disk. This value was determined by running a Connectathon write test against a local disk. We also know the throughput achieved in loopback from prosecuted tests.

To determine T_{op}, a value for throughput in loopback must be found. The result, as before, should be as unbiased by NFS caching as possible, and expose raw performance. Using the values from Table 9.3 different values for throughput in loopback mode were determined for different disk speeds. A

throughput in loopback mode for Beast was found to be 275.53 Mbps, and for Revanche it was to be 250.20 Mbps the difference being again due to the different disk I/O speeds noted in Table 9.3.

After finding the throughput in loopback, T_{op} is simply the difference between T_B and the amount of time to process one 8KB block, as presented in equation 9.5. This yields a T_{op} of 61.87 μs for crossover connectivity, and 24.55 μs for switched connectivity. The contribution of T_{op} to throughput in loopback mode is 26.01%, and 9.37% for crossover and switched connectivity respectively. The equation for this calculation is shown in 9.5.

The value of T_{op} should be equal regardless of network connectivity because T_{op} takes into account the processing time for NFS, RPC, UDP, IP, NIC, and NIC driver. The value of T_{op} used in the rest of our calculations was 61.87 μs as it proved to produce more accurate results when compared against actual measured performance.

$$T_{op} = \frac{1}{Throughput} - T_B \tag{9.4}$$

$$Contribution\ of\ T_{op} = \frac{T_{op}}{T_{op} + T_B} \tag{9.5}$$

Recall that FSB changes had a minimal effect on the performance of NFS, even though the frequency was changed by 50% from 90 MHz to 133 MHz. In fact a performance increase of 14% was observed. Taking into the account the contribution of T_{op}, a predicted increase in throughput over the FSB range is 11.5%. Recall that T_{op} is directly related to the processing power of the machine, and is the only value in loopback that is affected.

9.5.4 Revisiting Host-to-Host Throughput

After determining T_{op}, the host-to-host predicted performance values were calculated again taking T_{op} into account. The results can be seen in Table 9.4.

Substituting in the value of T_{op} (solved for in loopback mode) greatly increased the accuracy of our calculated throughput compared to the measured throughput of NFS with a single nfsd. The accuracy varied from 0.51% to 3.15%. Our phenomenological model has been proven to accurately describe the performance of NFS under various conditions.

Connection	$T_B(\mu s)$	Estimated Throughput (Mbps) $T_{op} = 61.87\mu s$	Measured Throughput (Mbps)	Accuracy
Switched	242.23	117.56	119.50	1.65%
Crossover	242.23	121.32	125.03	3.06%
Switched	175.98	132.11	128.08	3.15%
Crossover	175.98	136.88	137.58	0.51%

Table 9.4: host-to-host latency NFS performance calculations for a single nfsd.

9.6 NFS vs. Processor Load

To some extent this model seems to contradict the results in Table 5.2 that indicates a strong relationship between processor load (as induced by MTU processing) and throughput. To verify that the performance of NFS is not heavily effected by processor load, a simple program was written that needlessly uses processing cycles. The processes was set to a *nice* level of -20 to give near real-time scheduling. A listing of this program can be found in Appendix B.14. During its execution NFS benchmarks were run to determine the program's effect on NFS throughput. The results can be seen if Fig. 9.7.

As predicted NFS performance suffered very little as a result of adjusting processor load, Fig. 9.7 illustrates this clearly. It would appear that the effect of a small MTU is not primarily in its increase in processing load imposed, but rather in the delay it imposes on the final dispatch of a packet. That is, the loss of performance was not due to an increase in the amount of time needed to generate packets due to processor load. Rather it was due to the increase in latency caused by the marshaling of a NFS block in the form of many MTUs with associated packet overhead.

9.7 NFS's Network Pace

The performance of Linux's NFS implementation is apparently limited by its ineffectiveness in obtaining the advantage of concurrent requests in which disk writes, network processing, and propagation take place in parallel. The model of host-to-host performance shown in Fig. 9.2 breaks down the costs associated with network latency, disk latency, and protocol processing. We have shown that the network has the ability to sustain high throughput to

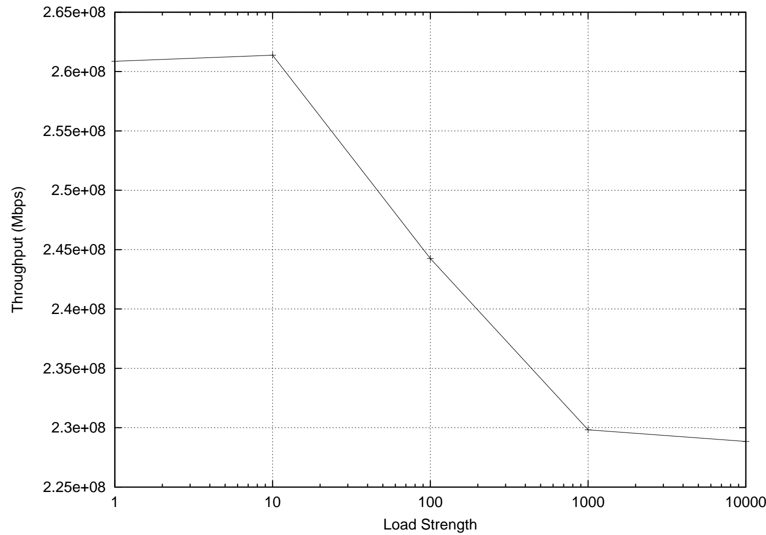

Figure 9.7: Processor load's effect on NFS performance.

quickly move data, and that the hard disk has the ability to process more data than it currently does. Clearly, multiple NFS requests (RPCs) could be acted upon concurrently by the server to increase the throughput of an NFS connection to that of a local file transfer. To understand the behavior of Linux's NFS implementation a packet trace of an NFS session during which a test was conducted was captured.

A Connectathon write test with ten unique 32 MB files written sequentially was executed, and the network traffic created as a result was captured with the packet sniffer Ethereal[2]. The test was conducted between Revanche and Beast, and eight nfsds were used on the server. Jumbo frames were used between Beast and Revanche for the test to make the packet trace easier to understand by removing the large number of IP fragments that would be generated as a result of using a smaller MTU. Table 9.5 shows a sample of this packet trace when the test was executing NFS write operations.

The "info" field of Table 9.5 shows the type of NFS operation that occurred between Revanche and Beast. Notice that an NFS write call is generated by the client (Revanche), and then acknowledged by the server (Beast) with an NFS reply. The interesting thing about this packet trace is that

[2]http://www.ethereal.com

No.	Time (μs)	Client	Server	Info
72816	19.653457	Revanche	Beast	V3 WRITE Call
72817	19.653818	Beast	Revanche	V3 WRITE Reply
72818	19.653852	Revanche	Beast	V3 WRITE Call
72819	19.654211	Beast	Revanche	V3 WRITE Reply
72820	19.654245	Revanche	Beast	V3 WRITE Call
72821	19.654607	Beast	Revanche	V3 WRITE Reply
72822	19.654641	Revanche	Beast	V3 WRITE Call
72823	19.655004	Beast	Revanche	V3 WRITE Reply
72824	19.655037	Revanche	Beast	V3 WRITE Call
72825	19.655398	Beast	Revanche	V3 WRITE Reply

Table 9.5: Packet trace of a Connectathon write test.

NFS transactions have developed a pace. The time between a NFS write call and write reply varies from 359 to 363 μs, and the time between NFS write reply and write call varies from 33 to 34 μs. NFS has fallen into a routine of waiting approximately 34 μs to acknowledge a write call, and then taking approximately 360 μs to execute a write call. What NFS should be doing is generating many outstanding NFS write calls, to increase throughput and efficiency because the network is able to handle the additional load, and the hard disk is idle during a majority of this cycle.

From the perspective of the agreement of our phenomenological model based upon sequential (non-concurrent) write operations and this packet trace, it would appear that this implementation of NFS does not implement concurrency properly. However, the $T_{cycle} \approx 390\mu s$ observed in this trace is also considerably less than the $T_{cycle} \approx 480\mu s$ computed with the phenomenological model. Thus while finding support for our model here we also find a contradiction with it.

In the next chapter we expand our analysis of the packet stream to discover the true mechanism behind the observed performance of NFS. We will show why our phenomenological model correctly describes performance while not literally describing the mechanism.

Chapter 10

Concurrency Observed

The phenomenological model of NFS presented in Chapter 9 demonstrates that the performance between a client and server is bounded by the ineffective use of concurrent transfers by NFS. This was predicted with our model, but we also observed this phenomenon by capturing an NFS session packet trace between Revanche and Beast. A Connectathon write test was conducted between Revanche and Beast with ten unique 32MB files written sequentially. The beginning of the NFS session specifically related to write calls and replies was recorded. The data are presented in Table 10.1.

The interesting feature of this trace is that NFS does in fact use concurrent transfers. In Table 10.1 sequence numbers 120, and 121 are sent within 55 μs of one another corresponding to two concurrent write calls. The reply for the first write call is sequence number 122, and the reply for the second write call is sequence number 123.

In this chapter we will abrogate the apparent contradiction between this observation and the success of our phenomenological model obtained in the last chapter.

10.1 Multiple NFS Servers and Clients

Because the first set of NFS write calls are concurrent it is reasonable to assume that the sequential accesses thereafter are not due to some limitation of the client, but rather they are due to the server. As the client is launching concurrent transfers to increase performance by filling the latencies associated with the network, and ensuring that RPC transactions are

No.	Time (μs)	Client	Server	Info
120	10.241409	172.16.1.2	172.16.1.1	V3 WRITE Call
121	10.241464	172.16.1.2	172.16.1.1	V3 WRITE Call
122	10.241863	172.16.1.1	172.16.1.2	V3 WRITE Reply
123	10.241865	172.16.1.1	172.16.1.2	V3 WRITE Reply
124	10.241921	172.16.1.2	172.16.1.1	V3 WRITE Call
125	10.241935	172.16.1.2	172.16.1.1	V3 WRITE Call
126	10.242281	172.16.1.1	172.16.1.2	V3 WRITE Reply
127	10.242333	172.16.1.2	172.16.1.1	V3 WRITE Call
128	10.242379	172.16.1.1	172.16.1.2	V3 WRITE Reply
129	10.242521	172.16.1.2	172.16.1.1	V3 WRITE Call
130	10.242665	172.16.1.1	172.16.1.2	V3 WRITE Reply
131	10.242734	172.16.1.2	172.16.1.1	V3 WRITE Call
132	10.242877	172.16.1.1	172.16.1.2	V3 WRITE Reply
133	10.242938	172.16.1.2	172.16.1.1	V3 WRITE Call
134	10.243081	172.16.1.1	172.16.1.2	V3 WRITE Reply
135	10.243131	172.16.1.2	172.16.1.1	V3 WRITE Call
136	10.243308	172.16.1.1	172.16.1.2	V3 WRITE Reply
137	10.243348	172.16.1.2	172.16.1.1	V3 WRITE Call
138	10.243433	172.16.1.1	172.16.1.2	V3 WRITE Reply
139	10.243494	172.16.1.2	172.16.1.1	V3 WRITE Call
140	10.243708	172.16.1.1	172.16.1.2	V3 WRITE Reply
141	10.243760	172.16.1.2	172.16.1.1	V3 WRITE Call
142	10.243845	172.16.1.1	172.16.1.2	V3 WRITE Reply
143	10.243905	172.16.1.2	172.16.1.1	V3 WRITE Call
144	10.244144	172.16.1.1	172.16.1.2	V3 WRITE Reply

Table 10.1: Packet trace of Connectathon write test between Revanche and Beast.

constantly available, the server appears to stall the client's concurrent behavior and force it into a sequential transfer mode. Evidence for this can be obtained by utilizing two clients with a server, and a server with two clients, and comparing the throughput achieved in each case.

10.2 Multiple NFS Servers

The first configuration involves use of Hutt as an NFS client, with Beast and Revanche as NFS servers. A Connectathon write test was conducted between Hutt, Revanche and Beast with ten unique 32MB files written sequentially. The throughput achieved between Hutt and Revanche, and Hutt and Beast is presented in Table 10.2. Recall that Beast has an hdparm measured performance of 42 MB/sec, and that Revanche has an hdparm measured performance of 23.11 MB/sec.

Client	Server	Throughput (Mbps)
Hutt	Revanche	81.58
Hutt	Beast	175.39

Table 10.2: NFS write test with Hutt as the client, and Beast and Revanche as the servers.

The data presented in Table 10.2 supports the claim that an NFS client can support concurrent transfers, but more data is needed to accurately confirm this. Repeating the same test with only a single connection between Hutt and Revanche, and between Hutt and Beast should have an observed throughput close to that observed in Table 10.2. The results for these test are shown in Table 10.3.

Client	Server	Throughput (Mbps)
Hutt	Revanche	93.88
Hutt	Beast	221.03

Table 10.3: NFS write test executed separately between Hutt as the client and Revanche as the server, and Hutt as the client and Beast as the server.

The result for an NFS write test executed only between Hutt and Revanche performed slightly greater (15%) than when Hutt was communicating

with two servers concurrently. The result for an NFS write test executed between Hutt and Beast performed greater still at 26%. This can be accounted for by the fact Hutt was able to receive enough data between Revanche and Beast concurrently that Hutt would have to wait on the completion of disk I/O before receiving additional NFS data.

10.3 Multiple NFS Clients

We believe the inability of NFS to use concurrent transfers is due to the NFS server implementation, and not the client. We have shown in Section 10.2 that a client connected simultaneously to two servers is able to achieve a throughput near the sum of the throughput of a client connected to each of the servers one at a time.

We have not yet shown that the performance of a server connected to two clients does not reflect the use of concurrent transfers. In this section, we use multiple clients accessing a single server. Beast and Revanche are now the clients, and Hutt is the server. We expect the data to show that both Beast and Revanche will have a decrease in throughput by half because the server does not effectively use concurrent transfers, and must now divide its throughput between two clients. A Connectathon write test was conducted between Revanche, Beast, and Hutt with ten unique 32MB files written sequentially. The data are presented in Table 10.4.

Client	Server	Throughput (Mbps)
Revanche	Hutt	53.18
Beast	Hutt	53.15

Table 10.4: NFS write test with Hutt as the server, and Beast and Revanche as the clients.

The data from Table 10.4 supports our claim. The fact that the measured throughput between Revanche and Hutt, and Beast and Hutt are within (0.06%) is in agreement with our previous data presented in Section 10.

Section 10 showed that after a few initial concurrent write calls, subsequent write calls were sequential. This would indicate that requests coming from two separate clients would be tightly interwoven with each other, and

that each client should get the same bandwidth assuming they are capable of sustaining the pace.

Chapter 11

NFS Packet Trace Analysis

Our phenomenological model is accurate when predicting the performance of NFS as a whole, but we have not observed the consistency of this model with individual packet traces, and where we see the performance of NFS on a per transaction basis. In this section, the packet sniffer Ethereal was used to capture test data between Beast and Revanche, and conversely, between Revanche and Beast. As a result we will finally reconcile all of our observations and the performance of our model. The test used for the packet traces wrote ten unique 32MB files sequentially to the NFS server. Jumbo frames were used during the test to make analysis of the packet trace easier.

11.1 Fast Disk Packet Trace

The first packet trace was taken between Revanche, and Beast. The trace data was used to construct two graphs. The first graph shown in Fig. 11.1 is a cumulative transfer plot. The graph tracks the amount of data transferred versus time. When an NFS write reply is received by the client, it signifies that an NFS write call has been accepted by the server. We record the time when an NFS write reply is received. Every NFS write reply received means that an 8KB NFS block was been transfered to the server. Therefore, we accumulate 8,192 bytes to the current running total. The total accumulated data up to the point of a received reply is recorded as a point on the graph. The slope of the plotted points indicates the throughput achieved by NFS.

Fig. 11.1 has two notable attributes. The first is the existence of two gaps, and the second is the change in slope seen after the first gap. The

gaps indicate that NFS was idle during periods of the tests. The gaps were examined more closely and it was observed that NFS commit operations were transmitted and RPC write requests timed out. (RPC timeouts were discussed in Section 8.1.)

No.	Time (seconds)	Info
25784	3.685857	V3 WRITE Call XID 0xa29a2094
25785	4.055445	V3 WRITE Call XID 0xa29a2094 dup XID 0xa29a2094
25786	4.065305	V3 WRITE Call XID 0xa39a2094 dup XID 0xa39a2094
25787	4.989262	V3 WRITE Reply XID 0xa29a2094
25788	5.465336	V3 WRITE Call XID 0xa39a2094 dup XID 0xa39a2094
25789	5.952341	V3 WRITE Reply XID 0xa29a2094

Table 11.1: NFS RPC write call retransmissions, and duplicates.

Table 11.1 shows the NFS write calls and replies that caused the first observed gap to appear in Fig. 11.1. The initial write call, number 25784 is made, and approximately 370 ms later it retransmits the same NFS write call. Another timeout is reached 379 ms from the original NFS write call, and another retransmission is sent. A reply for the original NFS write call is received, but apparently ignored because the NFS write call is sent yet again 1.779 seconds from the time of the original NFS write call. Finally, after 2.266 seconds from the original NFS write call, an NFS write reply is received and accepted. A total of 2.266 seconds were consumed in the processing of this single retransmission, and an additional NFS write call was sent when it was not required. This behavior was repeated for the second gap observed in Fig. 11.1.

We believe the cause of the gap can be attributed to NFS's having filled an internal buffer, and then having to commit this buffer to disk. We believe this because it is notable that 100MB worth of data was transfered before each gap at a rate too high to allow file write operations according to our model. Furthermore, the length of the gap corresponds with the amount of time it would take to flush the number of NFS transactions that occurred leading up to the gap. An NFS block is 8KB is size, and it requires 12,800 NFS transactions to transfer 100MB of data. Recall that the time required to write an 8KB NFS block using Beast's hard disk is 175.98 μs. This corresponds to a total time to write 12,800 NFS blocks of 2.25 seconds. This value is extremely close to the length of time of the first gap time of 2.266 seconds.

The first slope shown in Fig. 11.1 describes a transfer only throughput of 259.99 Mbps. The second slope of Fig. 11.1 describes a throughput of 133.85 Mbps, or approximately half of the throughput of the first slope. We believe the decrease in NFS transfer throughput performance is the result of the first disk commit disturbing the timing between the client and server. The disruption of this timing has caused the synchronization of NFS write calls and replies, that is the elimination of the concurrency present initially, thereby causing a decrease in NFS performance.

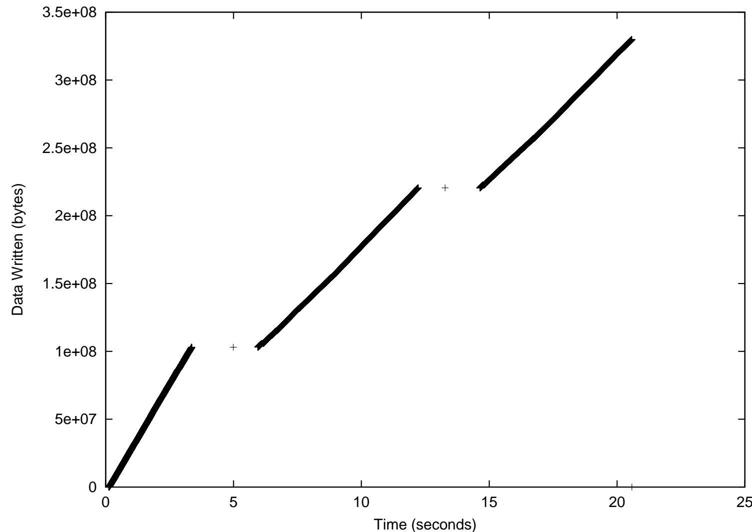

Figure 11.1: This plot shows the cumulative transfer versus time for a Connectathon write test between Revanche and Beast.

Another interesting fact of this data is the broad dispersion of points that occur before the first gap, and the distinct bands seen after the first gap. This indicates that after the first time data was committed to disk, transactions fell into a specific pattern for subsequent NFS transactions.

Fig. 11.2 plots the throughput achieved after an NFS transaction. A single transaction is composed of an NFS write call and an NFS write reply. Fig. 11.3 plots the reciprocal of Fig. 11.2 and shows the elapsed time between an NFS write call and an NFS write reply.

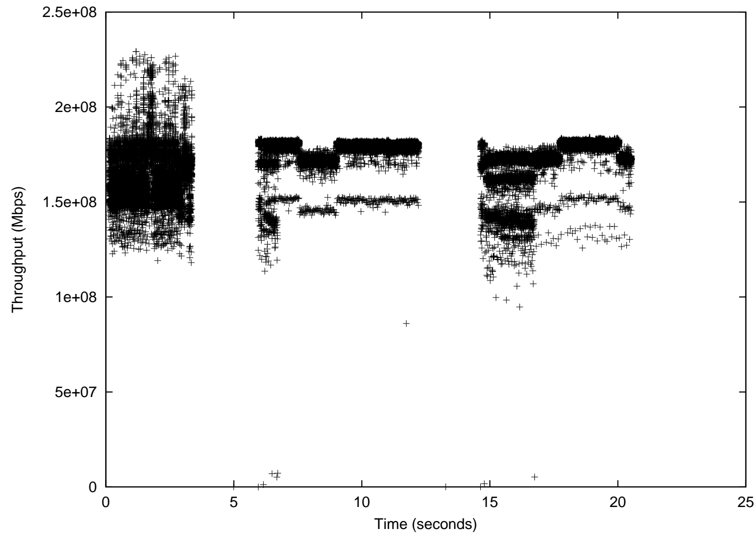

Figure 11.2: Per transaction throughput for write test described in Section 11.1.

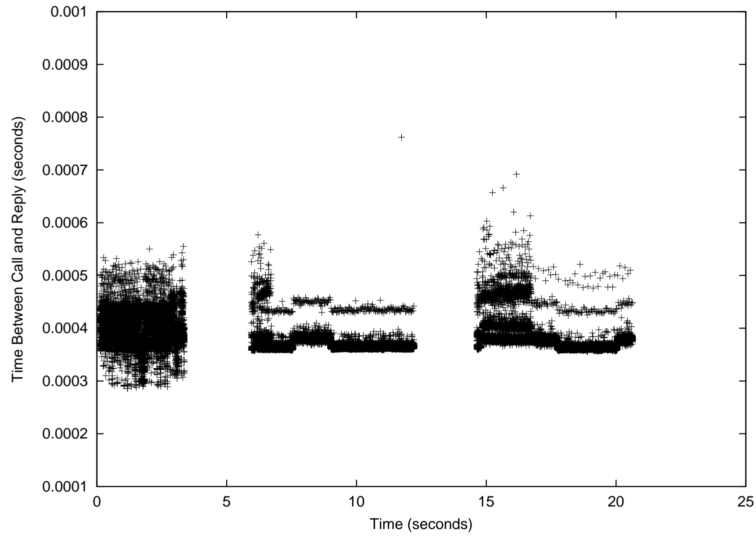

Figure 11.3: The time between write call requests and write call replies.

11.2 Slow Disk Packet Trace

Packet traces were next observed between Beast and Revanche to see the effect of a slower disk. Recall, that Revanche has an hdparm measured disk performance of 23.11 MB/sec, while Beast has an hdparm measured performance of 42 MB/sec. We would expect that the introduction of a slower disk should have affect the time to commit data to disk, and not the transfer only throughput achieved by NFS.

Fig. 11.4 shows the cumulative transfer graph, and the most recognizable feature of this graph is the large number of gaps that exist. The gap behavior was due to a number of retransmissions, except for the second half of the first gap starting at approximately 6.4 seconds, and extending to 8.5 seconds. During this large gap of time 24 UDP packets from the client on port 2049 were sent to the server on port 796. Port 2049 corresponds to NFS, and the functionality of port 796 cannot easily be determined because it was portmapped.

A portmapper[1] is a network program that runs on a well known port on the server. When a client wants to access a service on the server, it first connects to the portmapper, and asks for the service's port number. This frees the server from having to use well defined ports for all of its services.

The graph in Fig. 11.4 shows that our assumption is correct, the gap has increased in size, while the transfer throughput of NFS remains the same. The first gap appears after the same amount of data as seen in Fig. 11.1 has been transfered, 100MB. The first slope of Fig. 11.4 describes a transfer throughput of 268.25 Mbps.

The first gap of Fig. 11.4 is made up of two components, the first is the committing of data to disk, and second is interaction between the NFS server (port 2049), and NFS client (port 796) as previously described. This behavior between the server and client was found to consume 2.19 seconds, while the write time consumed 2.99 seconds. Using the write time, and the disk performance of the client (Revanche) one can verify the this gap corresponds to time required to commit the data to disk. Recall that the performance of Revanche's disk with a local disk Connectathon test was 242.23 μs to write an 8KB block of data. To write 12,800 NFS blocks, which account for 100MB worth of data requires 3.10 seconds, which is within 0.11 seconds of the actual time.

[1] *NFS Illustrated*, Brent Callaghan, pg 38

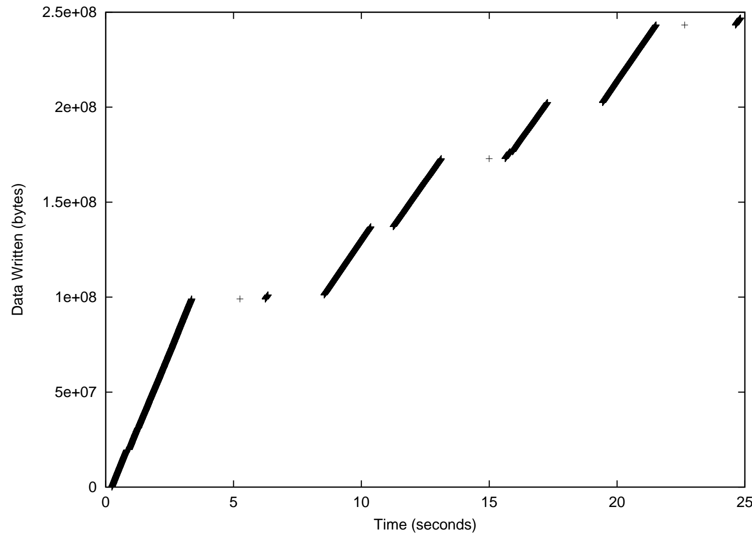

Figure 11.4: This plot shows the cumulative transfer versus time for a Connectathon write test between Beast and Revanche (slower hard disk).

The behavior exhibited in Fig. 11.4, Fig. 11.5, and Fig. 11.6 for the packet trace between Beast and Revanche is similar to that seen in the packet trace between Revanche and Beast. The exception being the larger number of retransmissions that occurred between Beast and Revanche. This was likely due to the slower disk on the server (23.11 MB/sec) versus the previous test's disk (42 MB/sec).

These graphs show that our model is not accurate on a per transaction basis, but rather it is accurate when considered over the aggregate of NFS transactions. NFS is able to use concurrent transactions to increase performance, but it is not able to commit data to disk concurrently. Furthermore, NFS transfers can fall into a nonconcurrent mode of behavior after interruption by some blocking mechanism such as the commitment mode. As a result of committing data to the disk network activity is halted thereby giving the appearance that NFS transactions are sequential. The performance of NFS can be improved by removing the need to stop network activity to commit data to the disk, and making disk and network activity for NFS concurrent.

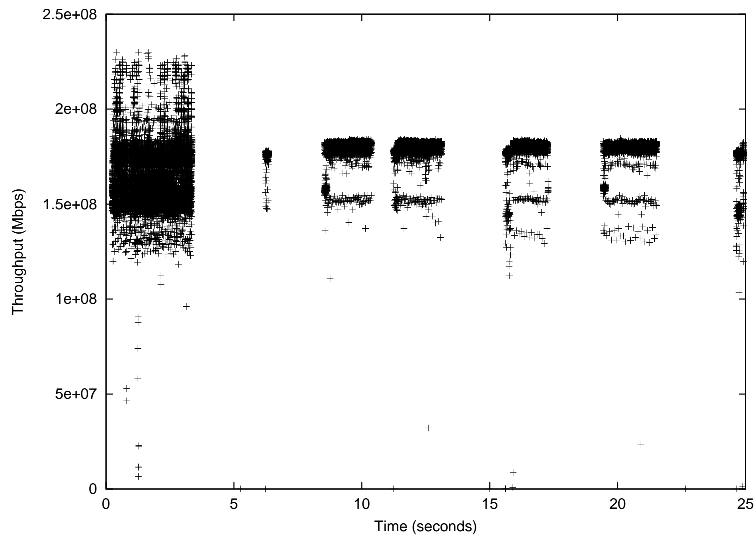

Figure 11.5: Per transaction throughput for test described in Section 11.1.

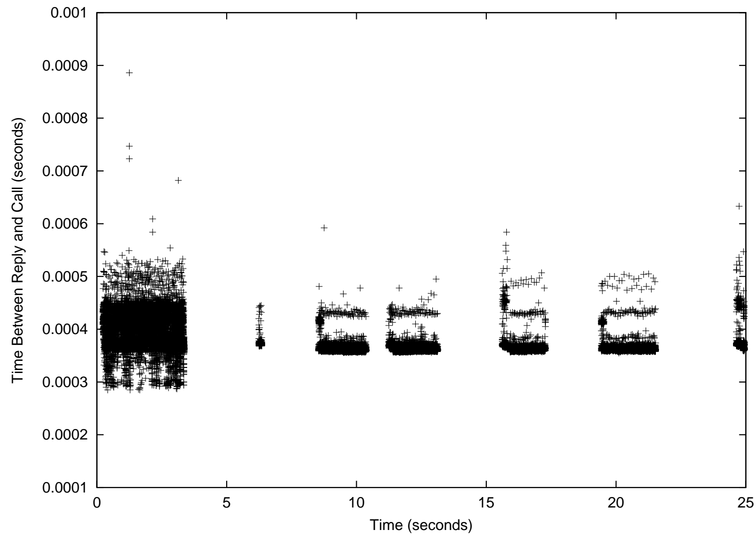

Figure 11.6: The time between write call requests and write call replies.

Chapter 12

Conclusion

This thesis presented the sequence of tests that were used to deduce the cause of poor write performance that we found in the current Linux NFS implementation. In the course of this investigation a variety of test scenarios were created to expose each facet of NFS performance from a black box perspective.

It was found that the performance of NFS is hindered by network latency, disk I/O speed, and NFS retransmission mechanisms. However, the performance which is achieved is far below that which is theoretically possible. The failure of NFS to perform at its optimum level and its sensitivity to network latency can be blamed upon an implementation that does not ensure the use of concurrency at the level that the overall protocol can support.

There is a performance ceiling due to processor and bus performance, and disk write speeds. We showed that both of these ceilings, with current technology, placed limits are well above the measured performance of NFS in host-to-host mode.

The ultimate causes for poor performance were two: NFS network data transfer in the test asynchronous mode are stopped during data commitment to disk, and data network transfers do not enjoy the full bandwidth that can be derived from maximum use of concurrency. Both of these problems should be addressed by implementors.

It should be possible to construct an NFS server that would:

1. implement concurrent NFS request handling and disk commitment;

2. implement a scheme that guarantees a sufficient number of concurrent network request-reply transactions so as to achieve the limit imposed

by either the disk write-speed or the network bandwidth.

Such an implementation would achieve NFS throughput comparable to direct disk access speeds for large file transfers. For example using our configuration, an NFS throughput over a switched network of 372 Mbps should be achievable in contrast to the 128 Mbps actually achieved.

12.1 Black Box Model

A black box model was developed that allows one to easily assess the performance of an NFS system with only a few measurable parameters. This is very useful because the proprietary nature of many implementations and devices. One need only know the speed of the network, the propagation delay of the network, and disk write performance to accurately determine an upper bound on NFS write performance for the tested implementation.

The black box model can be extended to account for more complex NFS systems. These NFS systems could include cascaded mount points wherein a file system is exported from one machine to another, and then exported again to another machine. It should also be possible to model NFS systems that make use of external RAID arrays connected via fiber channel.

12.2 Future Work

A new study should be done with NFS implementations other than the Linux based one that we used. Using the packet trace method a comparison should be made of NFS operations as implemented with BSD, Sun Solaris, and other major UNIX implementations. The data presented in this thesis only made use of asynchronous transfers, and a study focusing on synchronous transfers should also be conducted.

Finally, a study of the source code and a program trace should be used to further pinpoint origin of the observed behaviors. This project would lead to a revision of NFS aimed at implementation of the two essential repairs identified here.

Bibliography

[1] Alteon Networks. "Extended Frame Size for Next Generation Ethernets."
alteon.azlan.cz/whitepapers/Extended%20Ethernet%20Frames.pdf

[2] Hal Stern, Mike Eisler, Ricardo Labiaga. *NFS and NIS*. O'Reilly and Associates, Inc., Sebastol, CA, 2001

[3] Erez Zadok. *Linux NFS and Automounter Administration.* SYBEX Inc. San Francisco, CA, 2001

[4] Brent Callaghan. *NFS Illustrated.* Addison Wesley Longman, Inc. Reading, MA, 1999

[5] Gian-Paolo D. Musumeci, Mike Loukides. *System Performance Tuning*, 2nd Edition. O'Reilly and Associates, Inc., Sebastol, CA, 2001

[6] W. Richard Stevens. *TCP/IP Illustrated, Volume 1.* Addison Wesley. Boston, MA, 1994

[7] W. Richard Stevens. *TCP/IP Illustrated, Volume 2.* Addison Wesley. Boston, MA, 1995

[8] Rich Seifert. *Gigabit Ethernet: Technology and Applications for High-Speed LANs.* Addison Wesley. Boston, MA, 1998

[9] Marshall K. McKusick et al. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley. Boston, MA 1997

[10] Martin, R. P., and Culler D. E., NFS Sensitivity to High Performance Networks. *SIGMETRICS* 1999, pp 71-82.

[11] McKusick, M. K., Ganger, G. R., Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. *USENIX Annual Technical Conference*, June 1999, pp. 1-17.

[12] Lever, C., and Honeyman, P., Linux NFS Client Performance. *CITI Technical Report 01-12*, 2001.

[13] Khalidi, Y. A., and Thadani, M. N., An Efficient Zero-Copy I/O Framework for UNIX. *Sun Microsystem Laboratories, Inc., TR-95-93* May 1995.

[14] Loeb, M. L. et al. Gigabit Ethernet PCI Adapter Performance. *IEEE Network* March/April 2001.

[15] Martin, R. P., and Culler, D. E., NFS Sensitivity to High Performance Networks. *SIGMETRICS*, 1999.

[16] Postel, J., “Internet Protocol,” RFC 791, Sep, 1981.

[17] Sun Microsystems, Inc., “RPC: Remote Procedure Call Protocol specification: Version 2,” RFC 1057, Jan, 1988.

[18] Sun Microsystems, Inc., “NFS: Network File System Protocol Specification,” RFC 1049, Mar, 1989.

[19] Mogul, J. C., Deering, S. E., “Path MTU discovery,” RFC 1191, Nov, 1990.

Appendix A

Inventory

A.1 NFS Test Hosts

```
Legacy:
       400 MHz PII
       256 MB SDRAM
       Slackware v8.0
       Linux 2.4.17

Bastion:
       350 MHz PII
       256 MB SDRAM
       Debian v3.0, Woody
       Linux 2.4.17


Polaris:
       300 MHz PII
       256 MB SDRAM
       Slackware v8.0
       Linux 2.4.17

Hutt:
       Quad Processor Xeon, 700 MHz 2MB L2 Cache
```

```
        5.5 GB RAM
        Slackware v8.0
        Linux 2.4.17, 2.4.18

Polaris+:
        1.7 GHz PIV
        128 MB RDRAM
        Debian v3.0, Woody
        Linux 2.4.17

Revanche:
        1800+ XP
        256 MB DDR SDRAM
        Debian v3.0, Woody
        Linux 2.4.18

Beast:
        1800+ XP
        256 MB DDR SDRAM
        Debian v3.0, Woody
        Linux 2.4.18

Crash:
        266 MHz PII
        256 SDRAM
        FreeBSD v4.5

Kaboom:
        Dual Processor PIII, 1 GHz
        512 MB DDR SDRAM
        FreeBSD v4.5
        Debian v3.0, Woody
        Linux 2.4.17
```

Appendix B

Source Code

B.1 Parse Connecathon Log Files

```
#!/usr/bin/perl

##########################################################
#
# Christopher Boumenot
#
# Reads the output of Connectathon and prints them out
# in csv format.
#
##########################################################                       10


use strict;

my (@test1, @test2, @test3, @test4, @test5a);
my (@test5b, @test6, @test7, @test8, @test9);

while (<>) {

        if (/created (.*) levels deep in (.*) seconds/) {                        20
                push (@test1, $1);
        } elsif (/removed (.*) levels deep in (.*) seconds/) {
                push (@test2, $1);
        } elsif (/\d+ getcwd and stat calls in (.*) seconds/) {
```

```
                push (@test3, $1);
        } elsif (/\d+ chmods and stats on  files in (.*) seconds/) {
                push (@test4, $1);
        } elsif (/wrote \d+ byte file \d+ times in (.*) seconds/) {
                push (@test5a, $1);
        } elsif (/read \d+ byte file \d+ times in (.*)  seconds/) {          30
                push (@test5b, $1);
        } elsif (/\d+ entries read, \d+ files in (.*) seconds/) {
                push (@test6, $1);
        } elsif (/\d+ renames and links on \d+ files in (.*) seconds/) {
                push (@test7, $1);
        } elsif (/symlinks and readlinks on \d+ files in (.*)\s+seconds/) {
                push (@test8, $1);
        } elsif (/\d+ statfs calls in (.*) seconds/) {
                push (@test9, $1);
        }                                                                    40
}

for (my $i=0; $i<@test1; $i++) {
        print "$test1[$i],$test2[$i],$test3[$i],";
        print "$test4[$i],$test5a[$i],$test5b[$i],";
        print "$test6[$i],$test7[$i],$test8[$i],";
        print "$test9[$i]\n";
}
                                                                             50
```

B.2 Connectathon Write Shell Scripts

```
#!/usr/bin/perl

require 5.6.0;

use strict;
use IO::File;
use Getopt::Std;

our %opts;
                                                                        10
our $nfs_server = 'beast';
our $nfs_mnt    = '/mnt';
our $nfs_export = '/tmp';

MAIN: {
        getopts("huxdn:s:", \%opts) or die "Invalid option(s)\n";
        usage() if $opts{h};

        my $iter = defined $opts{n} ? $opts{n} : 1;
        my $filesize = defined $opts{s} ? $opts{s} : 1048576;          20

        my $testname = defined $opts{u} ? "test5c" : "test5a";

        if ($opts{d}) {
                for (my $i=0; $i<$iter; $i++) {
                        unlink("tg$i") or warn "$!\n";
                }
                exit(0);
        }
                                                                        30
        for (my $i=0; $i<$iter; $i++) {
                my $fname = sprintf("tg%d", $i);
                my $fh = IO::File->new($fname, "w");
                print $fh <<EOT;
\#!/bin/bash

export NFSTESTDIR=\"/mnt/revanche\.test/$i/"
\./$testname -t $opts{s}

EOT                                                                     40
                $fh->close;
        }
```

```
        if ($opts{x}) { exit(0); }

#       my $cmd = "mount -t nfs $nfs_server:$nfs_export $nfs_mnt";
#       print "$cmd \n";
#       system ($cmd);

        for (my $i=0; $i<$iter; $i++) {                                    50
                system("/bin/bash tg$i &");
        }

}


sub usage
{
        print "\nRead the source!\n";
        exit(1);                                                           60
}
```

B.3 Threaded Connectathon Write Test

```
#!/usr/bin/python

###########################################################
#
# Christopher Boumenot
#
# Multithreaded program were multiple threads are run
# against an NFS server.  Once all threads terminate
# the size is slowly increased, and the threads are
# run again.                                                   10
#
###########################################################

import threading
import time
import re
import os
import sys

NO_THREADS = 8                                                 20
MAX_SIZE   = 33554432
ITER_SIZE  = 16384
START_SIZE = 16384

class tee:
        def __init__(self, *fileobjects):
                self.fileobjects=fileobjects
        def write(self, string):
                for fileobject in self.fileobjects:
                        fileobject.write(string)               30

class writeNFS(threading.Thread):
        def __init__(self, num, size ):
                self.num = num
                self.size = size
                self.time = 0;
                threading.Thread.__init__(self)
        def run(self):
                output = os.popen('sh pg%d.sh %d' % (self.num, self.size))
                #print output.read()                           40
                match = re.search(r"\d+\.\d+", output.read())
                if match:
                        self.time = match.group(0)
```

```
    def getTime(self):
        return self.time

LogFile = open('incr_keep.log', 'w')
sys.stdout = tee(sys.stdout, LogFile)

for size in range(MAX_SIZE/ITER_SIZE − START_SIZE/ITER_SIZE):                    50
    actual = (size+1)*ITER_SIZE + START_SIZE
    print "File transfer size=", actual

    os.popen('mount -t nfs gpirus1.ece.wpi.edu:/tmp /mnt')

    for iter in range(NO_THREADS):
        t = writeNFS(iter, actual)
        t.start()

    ThisThread = threading.currentThread()                                        60
    while (threading.activeCount() > 1):
        CurrentActiveThreads = threading.enumerate()
        for WaitThread in CurrentActiveThreads:
            time.sleep(1)
            if (WaitThread != ThisThread):
                WaitThread.join()
                print WaitThread.getTime()

    print
    os.popen('umount /mnt')                                                       70
```

B.4 Ethereal NFS Packet Analyzer

```
#!/usr/bin/perl

#require 5.6.0;

use strict;
use IO::File;
use Getopt::Std;
use File::Basename;

use constant NFS_BLOCK_SIZE => 65536;  # bits                                    10

use vars qw (%opts %data $Gplot);

MAIN: {
    getopts('hf:g:', \%opts) or die "Unknown option $!\n";

    $Gplot = (defined $opts{g}) ? $opts{g} : "/usr/local/bin/gnuplot";

    usage() if $opts{h};
    usage() unless $opts{f};                                                      20

    my $fh = IO::File->new("< $opts{f}");

    while (my $line = $fh->getline()) {

        # skip comments
        if ($line =~ /^#/) { next; }

            my $seq;
            my $time;                                                             30
            my $info;

            if ($line =~ m/dup XID/) {
                print STDERR "found dup, ignoring\n";
                next;
            }

            if ($line =~ m/(\d+) (\d+\.\d{6}).*(V.*)/) {
                $seq  = $1;
                $time = $2;                                                       40
                $info = $3;
            }
```

```
    if ($info =~ m/WRITE.*XID (0x[a-f0-9]+)/) {
        my $key  = $1;

        if ($info =~ m/Reply/) {
            $data{$key}{reply} = $time;

            if (!defined $data{$key}{time}) {                    50
                $data{$key}{time} = $time;
            }

        } elsif ($info =~ m/Call/)  {
            $data{$key}{call} = $time;

            if (!defined $data{$key}{seq}) {
                $data{$key}{seq} = $seq;
            } else {
                die "non-unqiue XIDs!\n";                        60
            }
        }
    }
}

$fh->close;

my $data = 0;

#   my $ver = &version();                                        70

#   my $ctfh = IO::File->new("> ct-$ver.dat");
#   my $stfh = IO::File->new("> st-$ver.dat");
#   my $rtfh = IO::File->new("> rt-$ver.dat");

my $name = basename($opts{f}, ".txt");

my $ctfh = IO::File->new("> ct-$name.dat");
my $stfh = IO::File->new("> st-$name.dat");
my $rtfh = IO::File->new("> rt-$name.dat");                      80

print $ctfh "#\n# DO NOT EDIT, GENERATED AUTOMATICALLY FROM $opts{f}\n#\n\n";
print $stfh "#\n# DO NOT EDIT, GENERATED AUTOMATICALLY FROM $opts{f}\n#\n\n";
print $rtfh "#\n# DO NOT EDIT, GENERATED AUTOMATICALLY FROM $opts{f}\n#\n\n";

foreach my $val (sort {$data{$a}{time} <=> $data{$b}{time} } (keys %data)) {

    my $elap = $data{$val}{reply} - $data{$val}{call};
```

```
            my $rate = NFS_BLOCK_SIZE / $elap;
            $data += 8192;                                                    90

            print $ctfh "$data{$val}{time} $data\n";
            print $stfh "$data{$val}{time} $rate\n";
            print $rtfh "$data{$val}{time} $elap\n";

            #print "$cnt $data{$val}{reply} $data{$val}{call}\n";
        }

        # generate the gnuplot
        gen_plot("ct-$name");                                                 100
        gen_plot("st-$name");
        gen_plot("rt-$name");

        exit(0);
    }

sub usage
{
        print STDERR<<EOT;
        usage: $0 [OPTION] -f <filename>                                      110

    -g <program>  Specify the location of gnuplot, default is $Gplot
    -h            Help

EOT

        exit(0);

}
                                                                              120
sub version
{
    for (my $i=0; $i<1000; $i++) {
        my $name = sprintf ("%.3d", $i);
        unless (-f "ct-$name.dat") { return $name; }
    }

    die "Cannot find version < 1000\n";
}
                                                                              130
sub gen_plot
{
    my $fname  = shift;
```

```
    my $xlabel = "Time (seconds)";
    my $ylabel;
    my $title;

    if ($fname =~ /^ct/) {
        $title = "Cumulative Transfer";                                  140
        $ylabel = "Data Written (bytes)";
    } elsif ($fname =~ /^st/) {
        $title = "Sequential Transfer";
        $ylabel = "Throughput (Mbps)";
    } elsif ($fname =~ /^rt/) {
        $title = "Sequential Transfer";
        $ylabel = "Time Between Reply and Call (seconds)";
    } else {
        die "gen_plot(): don't know filetype $fname\n";
    }                                                                    150

    open (GNUPLOT, "|$Gplot") or
        die "Cannot open $Gplot\n";

    print GNUPLOT<<EOT;
#set title "$title"
    set xlabel "$xlabel"
        set ylabel "$ylabel"
#set term postscript eps
        set term postscript                                              160
        set data style points
        set output "$fname.eps"

        plot "$fname.dat" using 1:2 notitle
EOT

        close (GNUPLOT);
}
```

B.5 Parse Linux's /proc

```
#!/usr/bin/perl -w

########################################
##
## Christopher Boumenot
##
## Reads and parses /proc/net/snmp, and
## other if they are in a like format
##
########################################                              10

use strict;
use IO::File;
use Data::Dumper;
use Getopt::Std;

our $grp;
our %dat;
our %opts;
                                                                      20
MAIN: {
        getopts('chp', \%opts) or die $!;

        usage() if $opts{h};

        $Data::Dumper::Indent = 1;
        my $fh = IO::File->new("< /proc/net/snmp");

        if ( -f ".snmpdump" ) {
                use vars '%olddat';                                   30
                do '.snmpdump';
        }

        while(my $line = $fh->getline()) {
                my @fields;
                my @vals;

                if ($line =~ /(\w+):/) { $grp = $1; }
                $line =~ s/\w+://g;
                                                                      40
                @fields = split(/\s+/, $line);
```

```
        $line = $fh->getline();
        $line =~ s/\w+://g;

        @vals = split(/\s+/, $line);

        for (my $i=0; $i<scalar(@fields); $i++) {
            next if $fields[$i] =~ /^s*$/;
            $dat{$grp}{$fields[$i]} = $vals[$i];                50
            print "$fields[$i]=$vals[$i]\n" if $opts{p};
        }
    }
    $fh->close;

    write_dump();

    exit(0) if $opts{p};

    # form the difference of the two                            60
    if ($opts{c}) {
        foreach my $proto (sort keys %dat) {
            foreach my $val (sort keys %{$dat{$proto}}) {
            $dat{$proto}{$val} -= $olddat{$proto}{$val};
            }
        }
    }


    # print the stuff I care about;                             70
    foreach my $proto (sort keys %dat) {
        next if $proto =~ /icmp/i;
        #next if $proto =~ /tcp/i;

        print "------$proto-----\n";
        foreach my $val (sort keys %{$dat{$proto}}) {
            print "$val=";
            print "$dat{$proto}{$val}\n";
        }
    }                                                           80
}

sub write_dump
{
    my $dfile = IO::File->new("> .snmpdump");
    my $dump = Data::Dumper->new( [\%dat], ['*olddat'] )->Purity(1)->Dump;
    print $dfile $dump;
```

```
    $dfile->close;
}
```

B.6 UDP RTT Client

```
// Christopher Boumenot
//
// Measure the roundtrip time between two hosts using
// UDP
//
// Date: March 6, 2002

#include <iostream>
#include <unistd.h>

#include "ptimer.h"
#include "udpsock.h"

int main (int argc, char *argv[])
{
 //
 // UDP RTT Client
 //
  UDPSock udpSock;
  PTimer pTimer;

        udpSock.open(5900);
//      udpSock.setServ("130.215.16.86", 6000);
//      udpSock.setServ("172.16.1.1", 6000);
//      udpSock.setServ("130.215.17.177", 6000);
//      udpSock.setServ("127.0.0.1", 6000);
        for (int i=0; i<100; i++) {
                udpSock.setServ("130.215.16.86", 6000);
                pTimer.start();
                udpSock.snd();
                udpSock.rcv();
                pTimer.stop();
                cout << "time elapsed=" << pTimer.getElapsed() << endl;
        }

  return 0;
}
```

B.7 UDP RTT Server

```
// Christopher Boumenot
//
// Measure the roundtrip time between two hosts using
// UDP
//
// Date: March 6, 2002

#include <iostream>
#include <unistd.h>

#include "ptimer.h"
#include "udpsock.h"

int main (int argc, char *argv[])
{
  //
  // UDP RTT Server
  //

  UDPSock udpSock;
  PTimer pTimer;

        if (!udpSock.open(6000)) { exit(0); }
        for (;;) {
                udpSock.rcv();
                udpSock.snd();
        }

  return 0;
}
```

B.8 Precise Timer Class Definition

```
// Christopher Boumenot
//
// Measure the roundtrip time between two hosts using
// UDP
//
// Date: March 6, 2002

#ifndef _PTIMER_H_
#define _PTIMER_H_

#include <iostream>

#define MAX_TSC_T ~0x0ULL

typedef long long int tsc_t;

class PTimer {

 public:
  PTimer();
  ~PTimer() {};

  inline void start() {
    startVal = pfm_rdtsc();
  };
  inline void stop() {
    endVal = pfm_rdtsc();
  };
  double getElapsed();

  // Methods
 private:

  inline tsc_t pfm_rdtsc() {
    tsc_t x;
    __asm__ volatile (".byte 0x0f, 0x31"
                      : "=A" (x));
    return x;
```

```
    };

  void clkFreq();

  // Variables
 private:
  tsc_t startVal;
  tsc_t endVal;
  double elapsed;
  unsigned long cpuSpeed;

};

#endif
```

B.9 Precise Timer Class Implementation

```
// Christopher Boumenot
//
// Measure the roundtrip time between two hosts using
// UDP
//
// Date: March 6, 2002

#include <sys/time.h>
#include <unistd.h>

#include "ptimer.h"


PTimer::PTimer()
{
  clkFreq();
}

double PTimer::getElapsed()
{
  // check for wrap around
  if (endVal >= startVal) {
    elapsed = MAX_TSC_T - startVal + endVal;
  } else {
    elapsed = endVal - startVal;
  }

  return ( static_cast<double> (elapsed/cpuSpeed));
}

void PTimer::clkFreq()
{
  tsc_t tsc1, tsc2;
  struct timeval tv1, tv2;
  float seconds;

  gettimeofday(&tv1, NULL);
  tsc1 = pfm_rdtsc();
```

```
  /* Perhaps this delay is too short? */
  usleep (50000);
  gettimeofday(&tv2, NULL);
  tsc2 = pfm_rdtsc();
  seconds = (tv2.tv_sec + tv2.tv_usec * 1E-6)
    - (tv1.tv_sec + tv1.tv_usec * 1E-6);

  cpuSpeed = static_cast<unsigned long int>
               ((tsc2 - tsc1) / seconds);
}
```

B.10 UDP Socket Class Definition

```
// Christopher Boumenot
//
// Measure the roundtrip time between two hosts using
// UDP
//
// Date: March 6, 2002

#ifndef _UDPSOCK_H_
#define _UDPSOCK_H_

#include <iostream>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define  MAXMESG 10

class UDPSock {
public:
  UDPSock() {};
  ~UDPSock() {};

  bool open(int nPort=6000);
  bool close();
  void snd();
  int rcv();

  void setServ(const char ipAddr[16], const int nPort);

private:
  int sockfd;
  int port;

  struct sockaddr_in serv_addr;
  struct sockaddr_in my_addr;
};
```

#endif

B.11 UDP Socket Class Implementation

```
// Christopher Boumenot
//
// Measure the roundtrip time between two hosts using
// UDP
//
// Date: March 6, 2002


#include "udpsock.h"

bool UDPSock::open(int nPort=6000)
{
  if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    cout << "open(): Can't create socket" << endl;
    return false;
  }

   memset( (void *) &my_addr, sizeof(my_addr), 0);

   my_addr.sin_family       = AF_INET;
   my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
   my_addr.sin_port         = htons(nPort);

   if ( bind(sockfd, (struct sockaddr *) &my_addr,
                                    sizeof(my_addr)) < 0) {
     cout << "open(): Can't bind to socket" << endl;
     return false;
   }
   return true;
}


int UDPSock::rcv()
{
  int     n;
  char    mesg[MAXMESG];

  socklen_t servlen = sizeof(struct sockaddr);
```

```
  n = recvfrom(sockfd,
               mesg,
               MAXMESG,
               0,
               (struct sockaddr*)&serv_addr,
               &servlen);

  if (n < 0) {
    cout << "rcv(): didn't recieve anything" << endl;
    exit(1);
  }
  return n;
}


void UDPSock::snd()
{
  char mesg[] = "Y";
  socklen_t servlen = sizeof(serv_addr);
  int mesglen = sizeof(mesg);

  if ( sendto(sockfd,
              mesg,
              mesglen,
              0,
              (struct sockaddr*)&serv_addr,
              servlen) != mesglen) {

    cout << "snd():" << endl;
  }
}

void UDPSock::setServ(const char ipaddr[16], const int nPort)
{
  memset( (void *) &serv_addr, sizeof(serv_addr), 0);
  serv_addr.sin_family      = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(ipaddr);
  serv_addr.sin_port        = htons(nPort);
}
```

B.12 UDP RTT Makefile

```
# Christopher Boumenot
#
# Measure the roundtrip time between two hosts using
# UDP
#
# Date: March 6, 2002

TARGETS         = client server

CXX             = g++
CXXFLAGS        = -Wall -O2
#CXXFLAGS       = -Wall -O2 -pedantic

.cpp.o:         $(CXX) $(CXXFLAGS) -c $<

all:            $(TARGETS)
.SUFFIXES:      .c

ptimer.o:       ptimer.cpp

udpsock.o:      udpsock.cpp

client:         client.cpp ptimer.o udpsock.o
                $(CXX) $(CXXFLAGS) -o client $@.cpp ptimer.o udpsock.o

server:         server.cpp ptimer.o udpsock.o
                $(CXX) $(CXXFLAGS) -o server $@.cpp ptimer.o udpsock.o
clean:
        rm -f *.o $(TARGETS)
# DO NOT DELETE
```

B.13 Compute Average from UDP RTT Output

```perl
#!/usr/bin/perl -w

################################################
#
# Calculate the average from the output of
# UDP RTT
#
################################################


MAIN: {
	my @vals;

	while (<>) {
		$_ =~ s/\w+\s*\w+=//;
		$float = sprintf("%e", $_);
		push (@vals, $float);
	}

	my $total = 0;

	foreach (@vals) {
		$total += $_;
	}

	my @avgs = sort @vals;

	print "Min=$avgs[0], Avg=", $total/scalar(@vals), " Max=$avgs[-1]\n";
}
```

B.14 Increase Processor Load

```c
#include <stdio.h>
#include <signal.h>

double cnt=0;


void catch_ctrl_c(int signo)
{
        printf("cnt=%.lf\n", cnt);
        exit(1);
}

int main (int argc, char *argv[])
{
        double i,j;

        struct sigaction act;

        act.sa_handler = catch_ctrl_c;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;

        if (sigaction(SIGINT, &act, NULL) > 0) {
                perror("Could not install SIGINT signal handler");
        }

        /* 1e6 ~ 50% CPU load */
        /* 1e7 ~ 90% CPU load */

        for (i=0; i<1e12; i++) {
                for (j=0; j<1e9; j++) {
                        __asm__ ("nop");
                        cnt++;
                }
                usleep(1);
        }

        printf("cnt=%.lf\n", cnt);
```

```
        return 0;
}
```

Appendix C

Default Operating System Parameters

C.1 Linux Parameters

All units are expressed in bytes, except for parameters with time in the description, they are expressed in seconds.

C.1.1 TCP/IP

Buffer Allocation

- tcp_mem: 48128 48640 49152
- tcp_wmem: 4096 16384 131072
- tcp_rmem: 4096 87380 174760
- rmem_default: 65535
- wmem_default: 65535
- rmem_max: 65535
- wmem_max: 65535

IP Fragments

- ipfrag_high_thresh: 262144
- ipfrag_low_thresh: 196608
- ipfrag_time: 30

Appendix D

NFS Tests

D.1 Additional NFS Tests

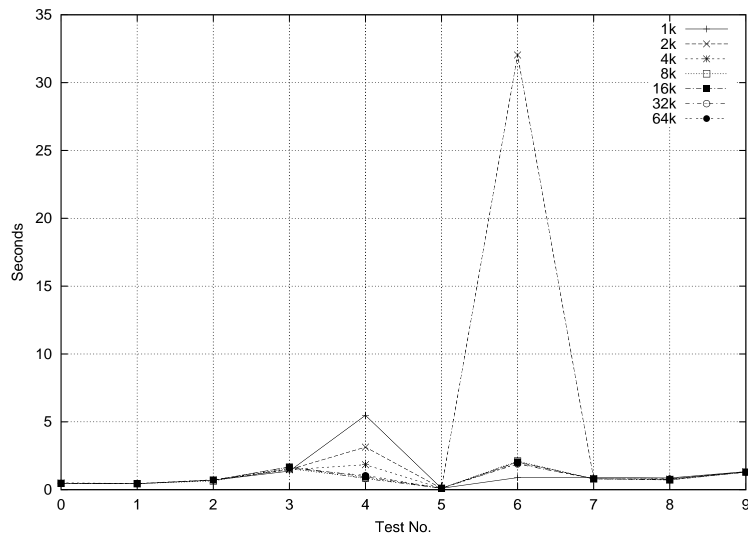

Figure D.1: Polaris+ to Hutt, varying block size