Achieving Scalable, Exhaustive Network Data Processing by Exploiting Parallelism

by

Afzal Mawji

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2004

© Afzal Mawji 2004

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Telecommunications companies (telcos) and Internet Service Providers (ISPs) monitor the traffic passing through their networks for the purposes of network evaluation and planning for future growth. Most monitoring techniques currently use a form of packet sampling. However, exhaustive monitoring is a preferable solution because it ensures accurate traffic characterization and also allows encoding operations, such as compression and encryption, to be performed.

To overcome the very high computational cost of exhaustive monitoring and encoding of data, this thesis suggests exploiting parallelism. By utilizing a parallel cluster in conjunction with load balancing techniques, a simulation is created to distribute the load across the parallel processors. It is shown that a very scalable system, capable of supporting a fairly high data rate can potentially be designed and implemented.

A complete system is then implemented in the form of a transparent Ethernet bridge, ensuring that the system can be deployed into a network without any change to the network. The system focuses its encoding efforts on obtaining the maximum compression rate and, to that end, utilizes the concept of streams, which attempts to separate data packets into individual flows that are correlated and whose redundancy can be removed through compression. Experiments show that compression rates are favourable and confirms good throughput rates and high scalability.

Acknowledgments

This research has been supported by a grant from Bell Canada.

I want to thank my advisor, Dr. Ajit Singh, for granting me the time and space necessary to carry out this research. He was patient when things were moving slowly, and excited when they were going well. He provided the idea for this research, the large computing resources required for it, and guided me through its completion.

I would also like to thank my readers, Professor Sagar Naik and Professor Sherman Shen.

I am indebted to Bernard Wong and David Sze. Bernard and I worked on the system implementation, and his ideas for the multiprocess version were extremely helpful. David provided invaluable technical support, particularly with his thorough knowledge of the FreeBSD operating system.

Tan Wang and Ngai-Pan Chow supported me when my work was not proceeding smoothly, and I hope that I was also able to help them with their work.

Finally, I want to thank Manisha Topiwala and my family. Their love, patience, and encouragement helped me through the tough times.

Contents

1 Introduction **1**

1.1 Exhaustive Monitoring 2

1.2 Load Balancing 3

1.3 Contribution 4

1.4 Outline of the Thesis 7

2 Background **9**

2.1 Network Monitoring 9

2.1.1 Active and Passive Monitoring 10

2.1.2 Packet Sampling 12

2.1.3 Traffic Flows 16

2.2 Exhaustive Monitoring 19

2.3 Using Specialized Hardware 22
2.4 Parallel Processing 23
2.4.1 Task Mapping 24
2.4.2 Exploiting Parallelism for Exhaustive Monitoring 28
2.5 Summary 29

3 Simulation with a Parallel Cluster 30

3.1 System Architecture 30
3.2 Implementation 32
3.3 Performance 34
3.3.1 Effects of CPU Activity 35
3.3.2 Effects of I/O Activity 40
3.3.3 Effects of Combined CPU and I/O Activity 44
3.4 Summary 47

4 Acquiring and Encoding Packets 49

4.1 Packet Capture and Release 49
4.2 Packet Encoding 52

4.3 Maintaining State with Streams 54
4.3.1 Stream Separation Factors 56
4.3.2 Mapping Web Traffic into Streams 59
4.4 Summary 60

5 System Model and Architecture 62

5.1 System Architecture 63
5.2 Load Balancing 65
5.3 Implementation 66
5.3.1 Combining Packets into Packages 67
5.3.2 Resynchronization 70
5.4 Performance 72
5.4.1 Compression Rates 73
5.4.2 Effects of CPU Activity 74
5.4.3 Effects of I/O Activity 78
5.4.4 Effects of Combined CPU and I/O Activity 81
5.5 Summary 84

6 Future Work and Conclusions **85**

6.1 Critical Assessment 85

6.1.1 Exhaustive Monitoring and Encoding 86

6.1.2 Transparency 87

6.1.3 Scalability 87

6.2 Future Work 88

6.2.1 I/O Performance 88

6.2.2 Load Balancing 89

6.2.3 Other Areas 91

6.3 Conclusions 91

Glossary **93**

List of Figures

2.1 Basic sampling algorithms (from [13]) 13

2.2 Layout of SMP and parallel cluster 25

3.1 Simulation system architecture 31

3.2 Throughput for CPU work 36

3.3 Throughput increase factor for CPU work 37

3.4 Throughput for I/O work 41

3.5 Throughput increase factor for I/O work 42

3.6 Throughput for combined CPU and I/O work 45

3.7 Throughput increase factor for combined CPU and I/O work . . 46

4.1 The tap virtual device 51

4.2 Stream separation scheme 58

5.1 Transparent bridge architecture 64

5.2 Throughput for CPU work 75

5.3 Throughput increase factor for CPU work 76

5.4 Throughput for I/O work 79

5.5 Throughput increase factor for I/O work 80

5.6 Throughput for combined CPU and I/O work 82

5.7 Throughput increase factor for combined CPU and I/O work . . 83

List of Tables

3.1 Throughput rates in Mbps for CPU work 36

3.2 CPU work throughput increase factor over two processors . . . 37

3.3 Throughput rates in Kbps for I/O work 41

3.4 I/O work throughput increase factor over two processors 42

3.5 Throughput rates in Kbps for combined CPU and I/O work . . . 45

3.6 Combined CPU and I/O work throughput increase factor over two processors 46

5.1 Achievable compression rates 73

5.2 Throughput rates in Mbps for CPU work 75

5.3 CPU work throughput increase factor over one processor 76

5.4 Throughput rates in Kbps for I/O work 79

5.5 I/O work throughput increase factor over one processor 80

5.6 Throughput rates in Kbps for combined CPU and I/O work . . . 82

5.7 Combined CPU and I/O work throughput increase factor over one processor 83

Chapter 1

Introduction

Telecommunications companies (telcos) and Internet Service Providers (ISPs) monitor traffic passing through their networks for the purposes of evaluating its performance and planning for future growth. These network operators would like to exhaustively monitor all packets to determine precisely how their networks are being used, but this is believed to be too difficult to achieve in real–time.

Considering the challenges involved in exhaustive, real–time processing of network data, most approaches in the past have resorted to data sampling techniques. While packet sampling may be satisfactory for certain monitoring purposes, it cannot be used for data encoding operations, such as com-

pression, encryption, content analysis for determining network usage, and packet filtering. In addition to requiring exhaustive monitoring, these operations tend to be computationally intensive and have proven difficult to achieve on current systems.

1.1 Exhaustive Monitoring

Unlike the sampling methods currently in use, exhaustive network monitoring allows data traffic to be characterized with 100% accuracy and makes it much easier to collect usage statistics at all network levels, from the data link layer right up to and including the application layer. As an added benefit, exhaustive monitoring would permit the possibility of encoding of the packets.

Packet encoding allows numerous operations to be performed on the data packets. For example, lossless compression may be used to increase the effective bandwidth of a link. Encryption would enhance the security of packets passing through untrusted networks. Packet filtering could be performed with many more degrees of freedom than the access control lists firewalls and routers currently permit. Packets could be filtered based on the contents of the packet, rather than just header information and both the header and

the actual content of the packets may be changed. Additionally, these packet operations may be combined, so that a packet can be both compressed and encrypted.

Exhaustive monitoring is highly desirable, but is generally not used because it is considered to be too computationally intensive to be practical. Packet filtering and encoding only add to the computational burden. Furthermore, these activities must be performed in real–time and must not reduce the network latency or throughput beyond a reasonable limit.

1.2 Load Balancing

It is vital that the monitoring and encoding operations be performed without adding significant delay to the network, but such a large amount of per–packet work would undoubtedly reduce the network throughput. To overcome this problem it is suggested that the monitoring and encoding be performed on a parallel processor since parallel processing is a cost–effective method for the fast solution of computationally large and data-intensive problems [27]. This would allow the monitoring system to scale easily to support higher throughput rates by simply providing additional processing capacity for its use.

Load balancing software would distribute the monitoring and encoding burden among the processors with each processor responsible for encoding one packet at a time. With a sufficiently large grain size for each CPU, it is expected that this would result in a speed–up as more processors are brought online. If the encoding process requires disk I/O then speed–up could be accomplished through parallel I/O, either in the form of multiple hard disks, or via RAID support.

1.3 Contribution

There are numerous challenges to overcome when building an exhaustive monitoring and encoding system. The system must be designed so that it does not disrupt the network when it is deployed, and ideally, the network should not need any reconfiguration whatsoever. The system must be relatively inexpensive and all components should be easily available. The system must perform online, real–time encoding, and make an attempt to maximize the utility of the encoding function. For example, in the case of data compression, the system should attempt to maximize the achievable compression ratio. In the event of packet loss or error, the system must be able to recover and resume operation. Finally, the system must scale well in order

to increase the achievable throughput.

This research investigates the feasibility of employing a parallel cluster along with load balancing techniques to develop an exhaustive data monitoring and packet encoding system. The data processing is performed in real-time by distributing the workload across the processors. In a typical setup, data arriving at the router would be passed on to the parallel machine for encoding before being sent off towards the destination. Ideally the network should experience no noticeable slowdown due to the encoding.

In this work, the actual data filtering operations performed on the data packets are abstracted as a certain amount of CPU and disk I/O operations. The objective is to determine the threshold at which encoding operations will reduce the throughput beyond reasonable limits. It is important to keep in mind that the goal of the system is to achieve the maximum possible throughput, not necessarily the most evenly distributed load.

A complete exhaustive monitoring system, performing encoding operations in real–time is implemented. The system supports any number of processors or hard disks and uses only off–the–shelf hardware and an open–source operating system. It captures packets from the network in a completely transparent fashion so that no network reconfiguration is necessary.

The system is scalable, meaning that it can handle increasing data traffic loads simply by adding more processors to the parallel machine, with no change to the software.

The data packets are divided into streams in an attempt to increase correlation between packets within the stream. The stream to which a packet should belong is determined from several factors, including the source and destination IP addresses, the URL of World Wide Web requests, the MIME or file type, and other intelligent heuristics. Separating the packets into streams increases the utility of many encoding functions, including data compression and encryption.

Upon completion of the encoding process, the packet is re–inserted into the network. In the case of error or packet loss on the link, a custom resynchronization protocol will automatically recover from such situations.

This thesis demonstrates that online, real–time, exhaustive monitoring and packet encoding is possible with off-the-shelf hardware and software. It is shown that increasing the number of processors and hard disks in the parallel machine allows CPU and I/O work to be completed in less time, thus permitting a higher throughput rate. Furthermore, for CPU–centric tasks, as the grain size is increased by increasing the amount of work performed

on each packet, adding more processors to the parallel machine improves the throughput increase factor.

1.4 Outline of the Thesis

Chapter 2 discusses network monitoring, covering the popular techniques of traffic sampling and flow monitoring. It then details the benefits of exhaustive monitoring and parallel processing, and explains why exhaustive monitoring is both a desirable and an achievable goal. The remaining chapters of this thesis describe the hardware and software architecture of the system in detail. Chapter 3 presents a simulation of the desired system to determine its feasibility. Using a parallel cluster, CPU and disk I/O operations are performed on the packets to help determine how well the system might scale. Chapter 4 describes how packets are transparently acquired from the network by the system, prepared for encoding or decoding, then encoded or decoded, and finally released back in to the network. Also covered is a crucial step in maximizing the utility of many encoding operations, the concept of data streams. Chapter 5 covers the complete implementation of the system, examining how well the throughput increases as the number of processors and hard disks increases, for both CPU and I/O work. Chapter 6

provides ideas for future research directions before drawing conclusions from this research.

Chapter 2

Background

This chapter presents two techniques that are used in network monitoring: packet sampling and traffic flow monitoring. It discusses the advantages of performing exhaustive monitoring, particularly using off–the–shelf components, and explains why exploiting parallel computing will allow the achievable throughput of an exhaustive monitoring system to increase with the use of more processors.

2.1 Network Monitoring

Monitoring network traffic serves several purposes. First, it allows the network operators to determine how the existing bandwidth is being used and

to evaluate whether or not the network is serving its current needs. Second, it allows them to allocate their resources more effectively. If a certain segment of the network is always heavily loaded while another consistently uses less than its capacity, the operator may decide to move resources to the heavily loaded network segment instead of purchasing new equipment. Finally, based on how the network is currently being used as well as on trends in usage, operators can plan for the future growth of the network, acquiring as much new capacity as is needed, and placing it only in the segments where it is required.

2.1.1 Active and Passive Monitoring

There are two broad categories of traffic measurement: active and passive monitoring. Active monitoring systems insert traffic, such as probe packets, into the network specifically to take a measurement [40]. Active monitoring more easily allows for measuring certain things such as network delay, path, and the loss rate. They typically use protocols such as the Internet Control Message Protocol (ICMP) and UDP or TCP packets. Luckie et al. propose a new protocol, the IP Measurement Protocol (IPMP), which is designed to overcome the drawbacks of the above–mentioned protocols because it was

created specifically with measurement in mind [34].

IPMP is based on an echo request and reply packet exchange scheme, similar to the *ping* application. However, it combines both the delay and path measurement mechanisms into a single packet exchange between the measurement host and the echo host, and is therefore provides more accuracy than separate measurements of these two metrics. An IPMP packet provides space for routers to insert path and time records, with the time record following the convention of the Network Time Protocol (NTP). In this manner, the complete path and delay times for each hop in the path can be determined at the measurement host.

Passive monitoring systems use only the traffic already existing on the network to take measurements. That is, they do not insert additional traffic into the network, but take all measurements from pre–existing traffic. Both active and passive measurement techniques require careful procedures, but other than how the packets are generated, are not really very different from one another according to Cleary et al. [14] and Deng [16]. The architecture presented in this thesis and those discussed in the remainder of this chapter, are passive monitoring systems.

2.1.2 Packet Sampling

A common practice in passive network monitoring systems is to sample the packets, because it is believed that monitoring each packet is not only computationally overwhelming, it is also impractical. Sampling involves making a trade–off between the benefit and the overhead of the measurements. The more frequently samples are taken, the more accurate the results will be. However, taking too many samples may overload the network monitoring system, and defeats the entire purpose of sampling.

The major difficulty with packet sampling lies in determining how many packets need to be sampled in order to ensure that the measurements are within a specific error tolerance. As the sampling granularity increases, the accuracy of the results decreases [15], indicating that a very high frequency of samples must be taken for high levels of accuracy.

Claffy et al. [13] present the three basic sampling algorithms: systematic, stratified random, and simple random. The systematic sampling technique involves dividing all packets into a series of n intervals, and deterministically sampling the kth packet from each of these n intervals. For example, if the network traffic is divided into n buckets, the 3rd packet from each bucket is selected. The stratified random algorithm also divides all packets into n

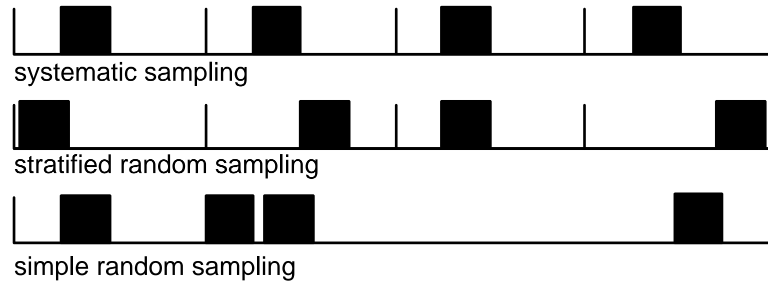


Figure 2.1: Basic sampling algorithms (from [13])

intervals, but instead selects a random packet from each of the n intervals. Finally, the simple random algorithm selects n random packets from the entire set. Figure 2.1 illustrates these three algorithms.

The sampling intervals used in these algorithms may be either time–triggered or packet–triggered. Time–triggered intervals are separated by a specific amount of time, whereas packet–triggered intervals are separated by a specific number of packets. For example, time–triggered intervals may be created every 5 seconds, while packet–triggered intervals may be created every 100 packets. Claffy et al. show that the time–triggered techniques perform more poorly than the packet–triggered ones, but that the performance differences are small. Furthermore, the performance differences in each of the three sampling algorithms, whether time– or packet–triggered, are also small.

There are many other sampling techniques that have been demonstrated

to be fairly effective, though they are usually extensions or variations on the three algorithms already mentioned. Three especially interesting proposals are: Cheng and Guang [7] suggest using Poisson sampling, particularly to determine the mean packet length; Guang et al. [29] propose a sampling method based on using the identification field of the IP header; and Erramilli and Wang [21] propose using fractal statistics to help describe traffic. There is also an IETF packet sampling working group named *psamp* whose charter is to define standards for packet sampling [1].

Static sampling methods, such as those described above, tend to produce inaccurate traffic measurements because they take all samples at the same rate, regardless of current traffic levels. During periods of low traffic, the network measurement processor has idle capacity but is still employing the same sampling granularity, and during periods of high traffic, the processor may be overloaded but attempts to continue sampling at a rate that it cannot keep up with.

Adaptive sampling adjusts the sampling granularity based on the CPU utilization, so that as the traffic and CPU usage increase, the sampling interval increases accordingly. Drobisz and Christensen [17] show that the adaptive sampling method produces more accurate traffic measurement than a

static sampling method. Choi et al. [9, 11] propose an adaptive random sampling technique that attempts to determine the optimum sampling interval according to traffic dynamics while bounding the sampling error within a pre–specified tolerance level.

The Simple Network Management Protocol (SNMP)[6] is a very popular passive monitoring system that collects measurements over intervals usually set to five minutes. Erramilli and Wang [21] suggest that due to the bursty nature of packet traffic, large time scales on the order of several minutes are too coarse to accurately indicate resource usage. These time scales have been carried over from circuit switched networks, even though packet switched traffic behaviour is fundamentally different.

Such large intervals serve as poor triggers for congestion controls, and may even conceal short–lived events of interest, such as performance degradation due to micro–congestion events. Papagiannaki et al. [39] state that these events may only last for times on the order of milliseconds, but they may still be of interest to some network operators.

2.1.3 Traffic Flows

Traffic flows are sets of packets that share some commonality, and based on certain common features, they are categorized into specific traffic types from which usage information is determined. Traffic flows are composed of sampled packets and therefore must also estimate the statistical properties of the original packet stream [19]. The IETF maintained a Real–time Flow Measurement (RTFM) working group, no longer active, whose charter was, in part, to produce an improved traffic flow model [2].

Many routers provide support for flows by keeping a table in memory with packet and byte counters as well as the times of the first and most recent packet arrivals for each flow. The flows are usually distinguished by source and destination IP address, and TCP or UDP port numbers [20]. Sommer and Feldmann [43] determine that NetFlow, Cisco's flow monitoring solution, can provide useful information, but that it requires great care and tends not to be as accurate as SNMP.

Claffy et al. [12] suggest using a richer set of parameters to characterize packets based on various locality conditions, both temporal and spatial, and assigning packets with matching parameters to the same flow. The first parameter proposed is based on flow timeout. All traffic passing between a

single source and destination pair, regardless of application type, is counted as a single flow as long as a given timeout interval is not exceeded between packets. The second parameter is based on traffic aggregation. It places all traffic based on certain information, such as destination network, destination host, source host, network pair, or host pair, into a single flow. For example, all traffic destined for a given network would be placed into the same flow, regardless of when the packets arrive or their type. The third parameter is based on transport and application layer information, such as the transport protocol or the application type. All packets with matching protocol or type information are placed into the same flow, regardless of the arrival time or source or destination addresses. The best solution would probably allow for a combination of these parameters.

Estan and Varghese [23] believe that keeping track of all flows is too expensive, and instead prefer to focus on a small number of flows that tend to account for a very large percentage of the traffic, dubbed elephants, stating that this is sufficient for many purposes. The main problem is how to identify the elephants, because tracking the volume of every flow defeats the entire purpose. Estan and Varghese focus on memory utilization and present two algorithms that identify the elephant flows, one providing implementation simplicity and the other providing greater accuracy. Both algorithms use a

small amount of memory, make a constant number of memory references per packet, and provide greater accuracy for the same amount of memory than most other sampling methods.

There are numerous other flow–based sampling algorithms, such as the one by Choi et al. [10], who propose an adaptive, stratified random packet sampling technique for flow–level traffic measurement. They demonstrate that the proposed sampling technique provides unbiased estimation of flow size with a controllable error bound, in terms of both packet and byte counts for the elephant flows, while avoiding excessive oversampling.

Cho et al. [8] believe that a major weakness of flows is that predefined filter rules are required to identify traffic types. The filter rules classify packets by examining fields in the packet header and thus require *a priori* knowledge of traffic types. Therefore, the rules cannot be used with applications that make use of dynamic ports and also provide limited usefulness in identifying new protocols and applications. Furthermore, there are far too many combinations of flow categories, and so in measuring only elephant flows, minor traffic types, which may grow with time, are ignored.

Estan et al. [22] state that flows provide insufficient dimensionality, tending to only support a few categories, like source address, destination address,

protocol, source port, and destination port. Furthermore, the predefined filter rules are fixed and do not support dynamic combinations of these categories. They present a method to automatically classify traffic into appropriate multi–dimensional clusters, with the clusters themselves defined automatically when they provide a meaningful aggregate of flows. Their approach still uses the same few categories, but combines and varies them dynamically. This allows new traffic patterns such as network worms to be classified automatically.

The concept of categorizing traffic into flows is a very useful one, and by combining it with exhaustive monitoring, very detailed network information can be determined. As explained in Chapter 4, these categories can also be used to aid in encoding data.

2.2 Exhaustive Monitoring

Packet sampling is useful in many applications, but it cannot provide perfect accuracy. The best it can do is provide a statistical error bound, which may not be sufficient in many circumstances. Inaccurate sampling can lead to incorrect decisions by network operators, which could be disastrous. Exhaustive monitoring provides perfectly accurate measurement and furthermore,

makes it possible to perform operations, such as compression or encryption, on the data packets.

Most monitoring systems only measure data at the data–link, network, and transport layers, ignoring the application layer completely. Feng et al. [24] state that additional insight can be obtained by monitoring and measuring traffic at the application level, something that is very difficult with packet sampling. Some applications use a specific port, and sampling can be used to monitor these types of applications, but other applications use dynamic ports and present application information only at the beginning of the connection. If these packets are not sampled, the information is lost to sampling systems.

Feng et al. present a system that monitors network traffic at the application level, but their solution requires a custom–modified Linux kernel and must be run on the computer that generated the traffic in the first place so that the packets are measured before going out onto the network, meaning that their solution is not practical in most cases. The same level of insight, however, can be obtained with exhaustive monitoring.

Higher level monitoring allows the network operators to determine how the network is being used by applications such as the World Wide Web, email, file transfers, and numerous other network applications. With an accurate

picture of the traffic, operators can identify evolving trends, such as the recent rise of peer to peer applications, and streaming music and video.

On those network links that charge clients based on usage, it is extremely important to obtain an accurate usage amount. Duffield et al. [18] provide a technique that samples mainly from the tail end of the distribution because they say that IP flows tend to have heavily tailed packet and byte size distributions. However accurate their technique may be though, it can only provide statistical error bounds, not the perfect accuracy given by exhaustive monitoring.

A tremendous benefit of exhaustive monitoring is that it allows the encoding of data. During peak hours additional bandwidth can be created by encoding packets with lossless compression algorithms. Sensitive data that must travel over unknown networks before reaching its destination can be encrypted automatically by the network. Data packets can be filtered based on the packet contents, instead of only header information. Though exhaustive monitoring provides numerous benefits, it is not considered practical because of the high computational cost. This thesis presents a system, described in detail in Chapters 4 and 5, for exhaustive monitoring and encoding that makes use of parallel processing.

2.3 Using Specialized Hardware

Exhaustive monitoring requires high levels of accuracy and reliability, and one way to ensure this is to use custom hardware designed specifically for network monitoring [14, 31]. These FPGA and ASIC solutions are very expensive however, and it is preferable to use off–the–shelf hardware and software in order to reduce the cost and complexity of the system. Fraleigh et al. [25] show that such a system, if carefully constructed, is reliable enough to perform exhaustive monitoring. They use a PC running Linux with a large disk array and a SONET network interface to exhaustively capture packet traces at rates as high as OC–48 (2.5 Gbps).

The infrastructure presented in this thesis makes use of only off–the–shelf hardware and an open–source operating system, thus keeping the costs to a minimum. These components, in conjunction with the software presented in this thesis, allow the system to be deployed with no reconfiguration of the network.

2.4 Parallel Processing

Parallel processing is a cost–effective method for quickly solving computationally large and data–intensive problems [27]. Parallel computers are now available as inexpensive commodity multiprocessors and clusters, and modern operating systems provide the ability to take advantage of concurrency.

When each processing element (PE) in a parallel computer is able to execute a different program independently of the other PEs, it is referred to as a multiple instruction, multiple data (MIMD) computer. A variant of this is the single program, multiple data (SPMD) model, in which each PE runs a separate instance of the same program and operates on different data. By contrast, in a single instruction, multiple data (SIMD) computer, each PE runs exactly the same instruction at the same time as all of the other PEs, but on a different piece of data.

A symmetric multiprocessor (SMP) system has tightly coupled processors in which the memory is physically shared among all processors so that each PE has equal access to any memory segment. Therefore, the time taken by any PE to access any memory in the system is identical. The memory address space may be exclusive to a processor (local) or common to all PEs (global). To communicate, PEs usually alter the data in the common memory address

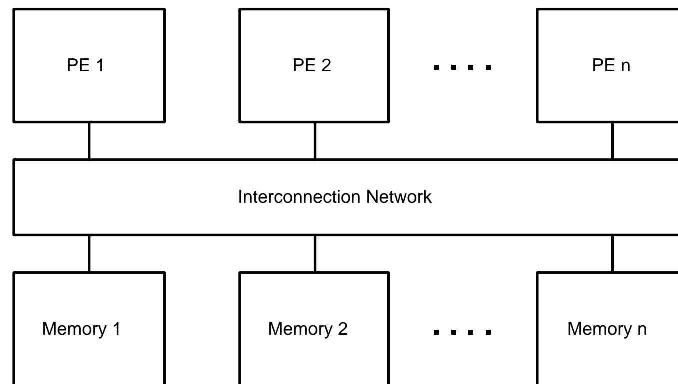
space. The processors and resources are all controlled by a single operating system.

A parallel cluster is a group of two or more computers, each running their own operating system, that are networked together so that they appear to applications as a single logical system. Each PE has only local memory and so PEs must communicate via message passing. Messages are used to transfer data, work results, and also to synchronize actions. Figure 2.2 illustrates the relationship between PEs and memory for both SMPs and clusters.

This thesis uses a parallel cluster for both simulation and implementation purposes. In both cases, each processor runs the same program, but not in lockstep, meaning that in each case, the design and implementation follow the SPMD model.

2.4.1 Task Mapping

In order to take advantage of concurrency, a problem must be divided into smaller tasks, which are then mapped onto processes, each of which is assigned to a CPU and executed in parallel. Introducing parallelism also brings about several sources of overhead. One type of overhead occurs immediately when the problem is divided into tasks because there must be some coordina-



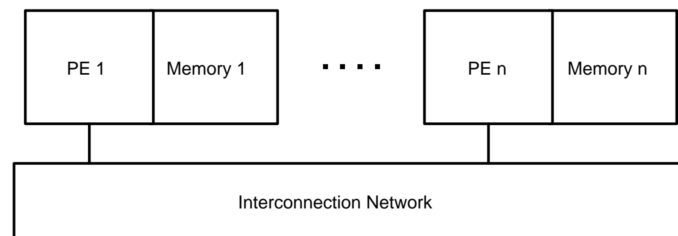


Figure 2.2: Layout of SMP and parallel cluster

tion among the processes when partitioning the data between them. During computation it is often necessary for processes to synchronize and possibly exchange data between processes in order to continue processing, a further source of overhead. Once the computation is complete, the data must be merged, resulting in another form of overhead. These types of communication between processes are referred to as interprocess communication (IPC). Another source of overhead is the time spent with processes idling, waiting for more data to process. Since the goal of concurrency is to minimize execution time, these overheads must be minimized.

A poor task mapping will result in an uneven load distribution, with some processes completing their tasks earlier than others. IPC and idling are both functions of the mapping and therefore determining a good mapping requires that the time spent doing both of these things be reduced. Unfortunately, trying to reduce one source of overhead tends to increase the other. For example, in order to reduce IPC, all tasks that need to communicate may be mapped onto a specific process, thus requiring virtually no IPC. This mapping will result in an unbalanced workload because all other processes will have little work to do, which increases the idling overhead. Taken to the extreme, it results in all tasks being mapped onto a single process, with the remaining processors being completely idle. To balance the load more evenly, it is usu-

ally necessary to assign tasks that require a high level of IPC onto different processes, resulting in an increase in IPC overhead.

Mapping techniques fall into two main categories: static and dynamic mapping. With static mapping, tasks are assigned to processes prior to execution. Choosing a good mapping depends on many things, such as the task sizes, the size of the data for each task, and the communication requirements Even with known task sizes, determining an optimal mapping is an NP–complete problem for non–uniform tasks [27].

In dynamic mapping, tasks are assigned to processes during execution. This is necessary when the tasks are generated dynamically, and is usually done if task sizes are unknown *a priori* because in such a case, a static mapping may produce a very unbalanced load distribution. It is usually more difficult to perform dynamic mapping than it is to perform static mapping.

When exploiting parallel computing, it must be kept in mind that there is a point where the overhead resulting from using many processes outweighs the speedup. The complexity of the synchronization among processes, or the amount of communication among processors may require so much overhead that the performance of the tasks themselves can be negatively impacted [30].

2.4.2 Exploiting Parallelism for Exhaustive Monitoring

This thesis suggests the use of parallelism to perform exhaustive monitoring. The main objection to exhaustive monitoring has been that it is too costly in terms of computation, but if the processing capacity of the monitoring system can by increased by employing a parallel processor, than it may be possible to exhaustively monitor packets in real–time.

Milward et al. [37] present a scalable, parallel multicompressor FPGA design that is capable of supporting rates of 100 Mbps using 10 compressors. However, as mentioned in Section 2.3, custom hardware solutions tend to be expensive and a solution using off–the–shelf hardware is preferable.

Mao et al. [35] use a parallel cluster to monitor and log HTTP traffic over a link. The cluster was able to handle up to 2000 concurrent HTTP connections with a bandwidth of nearly 50 Mbps without losing any packets, using 100 Mbps network connections between the machines in the cluster.

This thesis presents a novel approach to exhaustive data monitoring as well as online data encoding. It presents and develops a reliable, scalable, and efficient infrastructure to perform real–time encoding of data packets online, while maintaining high bandwidth rates. A parallel cluster is used to spread the computational load over several processors, allowing the moni-

toring system to scale upwards to support higher bandwidth rates by simply adding more processors to the system.

2.5 Summary

This chapter provided an overview of current network monitoring techniques, particularly packet sampling and traffic flows, and showed why exhaustive monitoring is a desirable alternative. To achieve the goal of exhaustive monitoring, it suggests the idea of using parallel computation to support high throughput rates that can easily scale upwards. The next chapter presents a simulation of an exhaustive monitoring system with a parallel cluster to determine how well the system might scale.

Chapter 3

Simulation with a Parallel Cluster

This chapter presents a simulation of an exhaustive monitoring and encoding system, and examines how well such a system might scale via parallel processing. The simulation is a system in which data traffic is encoded by a parallel cluster before being passed to the next hop. The performance of CPU and disk I/O operations with an increasing number of CPUs and hard disks is determined.

3.1 System Architecture

The simulation system is based on a load balancing master–slave architecture. This is necessary because only one machine is physically connected to

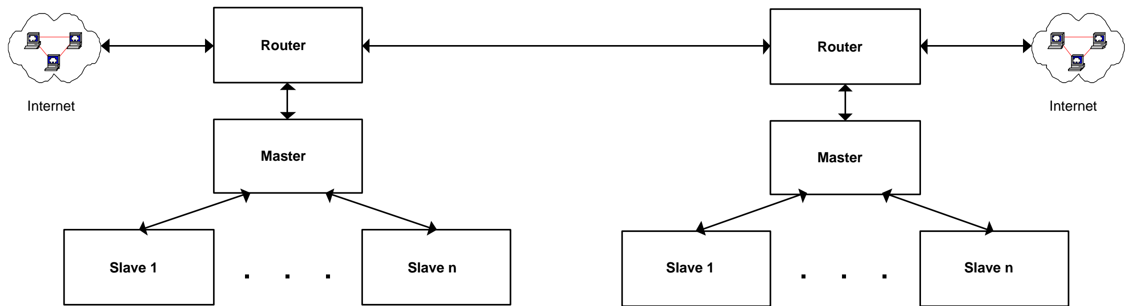


Figure 3.1: Simulation system architecture

the router. The router passes all incoming packets from the network to the master in the parallel cluster, and all packets received back from the master are forwarded as usual. This way, the router does not need to perform any computational work or load balancing itself. Figure 3.1 illustrates the system architecture.

The master is the machine within the parallel cluster that is linked to the router. It maintains a list of all slave processors and their current loads, and when it receives an incoming packet from the router, the master assigns it to the most lightly loaded slave. If the master has idle CPU capacity it will also perform operations on the packets instead of sending the work to a slave. Once the encoding is completed by a slave, it is sent back to the master, which then returns back to the router for forwarding. If there is an error when sending a packet to the slave, it is assumed to have become disconnected, is removed from the slave list, and the next best slave is chosen.

In this manner, slaves can disconnect at any time.

The master does not keep copies of the packets sent to the slaves, as this would put memory requirements beyond a practical level. Therefore, if a slave disconnects, all outstanding work it has been assigned is lost. In that case it is expected that higher layers of the network will take care of retransmission to ensure reliability.

Slaves simply wait for work from the master, perform the required computation, and then, if necessary, send the results back to the master. They can connect to or disconnect from the master at any time.

3.2 Implementation

Simple router, master, and slave programs were written as application–level software. As a result, this simulation does not deal with actual packets, but with buffers that are streamed via TCP. This means that the buffered data do not reflect actual packets and that there are no headers to aid in the processing. Any packets received by the router from the outside network are sent to the master for processing. Packets received back from the master are sent to the corresponding router, which then passes it on to its own parallel cluster before sending it to the outside network. The slave programs running on all

slave machines are identical.

In this simulation, we are not concerned with the specific encoding operations that are performed by the parallel cluster and so they are abstracted into units of CPU and disk I/O work. One unit of CPU work is defined as the CPU time required for compressing an incoming packet using LZO compression [38] on a slave processor. LZO is an open source compression algorithm based on the Lempel–Ziv scheme and provides very fast compression with fair compression rates. One unit of I/O work is defined as opening a new file handle, writing the uncompressed packet to the local disk, closing the file handle, re–opening it, and then reading back the file back. All disk writes are followed by a call to the *fsync* library function, which forces data to be written to the disk and not stored in cache memory for later reading or writing.

The system was implemented on a parallel cluster consisting of eight dual–processor Pentium III 500 MHz machines with 512 MB RAM running FreeBSD. All nodes in the network and within the cluster are connected via a 1 Gbps Ethernet switch. The code was written in C++ and is approximately 3200 lines.

3.3 Performance

Several experiments were run to determine the performance of the system and determine how well it scales as more processors and hard disks are added. In all of the experiments conducted, random data was sent to the sending–side router, which then passed the packets on to the sending–side parallel cluster. Depending on the particular goal of the experiment, certain amounts of CPU and disk I/O work were performed on the packets, which were then sent back to the router for forwarding. Once the packets were received at the receiving–side router, it too passed the packets to the receiving–side parallel cluster for processing before being sent on to the final destination. In this manner, packets were simultaneously sent in both directions across the network.

Each router has one master and up to three slave nodes in its parallel cluster. Since each machine is a dual processor, this means that between two and eight CPUs may be used for processing at either end. Each node of the cluster contains a single 10K RPM SCSI hard disk for I/O work.

The goal of the experiments is to maximize the throughput of the system under the encoding operations specified and is determined by examining how fast data that is inserted into one end of the system comes out of the other

end, that is, after it has passed through the entire system. These experiments are not attempting to determine the increase in achievable bandwidth through compression, so even though the data may be compressed by the parallel cluster, it is the uncompressed data that is forwarded to the next hop. Each experiment is run until the throughput value becomes stable.

3.3.1 Effects of CPU Activity

Table 3.1 and Figure 3.2 illustrate how the throughput (in Mbps) changes with the number of processors for the given CPU work units. In these experiments there is no I/O work being done, although the overhead for the associated function calls are still included. Table 3.2 and Figure 3.3 show how much increase in throughput is achieved by utilizing more processors in the cluster. The throughput increase factor is calculated as the throughput achieved with n processors divided by the throughput achieved with a single processor. It indicates how many times faster the throughput has become with multiple processors. Each column in Table 3.2 gives the increase factor over one processor, so for example, the first row indicates that adding additional processors actually reduces the throughput by about half as compared to using only a single processor from the cluster.

CPU Work Iterations	Number of Processors							
	1	2	3	4	5	6	7	8
0	170.02	66.29	92.52	83.46	86.94	85.40	83.57	82.81
1	94.02	81.05	84.28	80.67	82.61	81.91	80.56	80.59
2	66.26	66.05	78.22	78.07	79.23	78.88	78.42	77.62
4	42.60	57.83	68.08	70.37	72.30	72.80	73.21	73.62
8	24.48	34.99	51.80	56.45	61.00	62.49	65.09	66.20
16	13.15	20.14	34.64	38.53	43.88	45.81	50.45	53.16
32	6.84	10.56	19.64	22.90	25.92	27.51	30.23	34.52
64	3.48	5.29	10.16	12.50	14.19	15.39	16.76	19.33

Table 3.1: Throughput rates in Mbps for CPU work

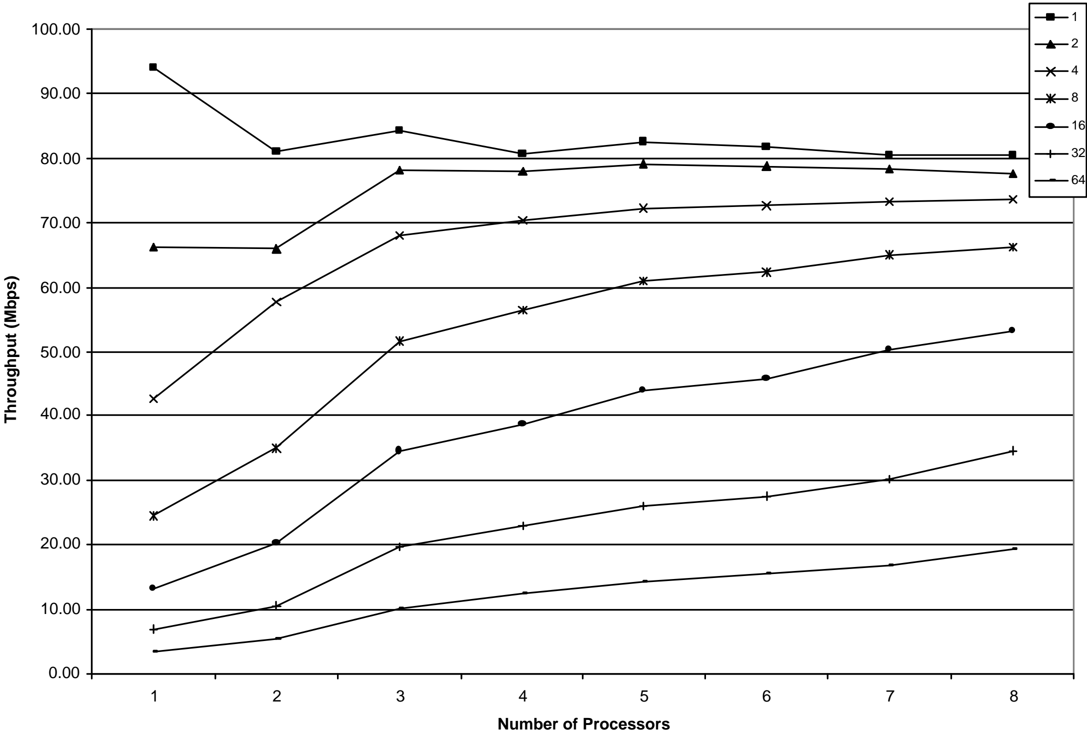


Figure 3.2: Throughput for CPU work

CPU Work Iterations	Throughput Increase Factor						
	2	3	4	5	6	7	8
0	0.39	0.54	0.49	0.51	0.50	0.49	0.49
1	0.86	0.90	0.86	0.88	0.87	0.86	0.86
2	1.00	1.18	1.18	1.12	1.19	1.18	1.17
4	1.36	1.60	1.65	1.70	1.71	1.72	1.73
8	1.43	2.12	2.31	2.49	2.55	2.66	2.70
16	1.53	2.63	2.93	3.34	3.48	3.84	4.04
32	1.54	2.87	3.35	3.79	4.02	4.42	5.05
64	1.52	2.92	3.60	4.08	4.43	4.82	5.56

Table 3.2: CPU work throughput increase factor over two processors

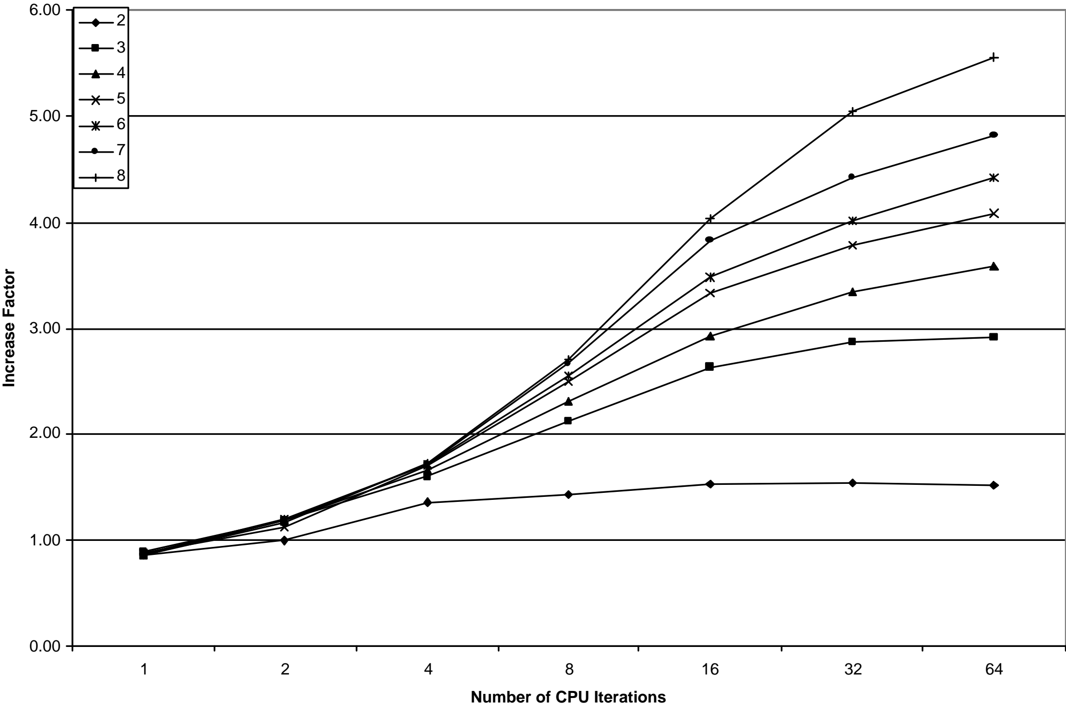


Figure 3.3: Throughput increase factor for CPU work

Tables 3.1 and 3.2 include data for zero CPU work units. This means that data passes through the entire system but absolutely no CPU or I/O work is done though all function call overheads are included. This number serves as an upper bound for the maximum throughput the simulation system can support. In this case, when the number of slave processors available to the master for load balancing increases, the maximum throughput actually decreases because we are introducing communication overhead but are not taking advantage of the extra processors by performing any encoding operations.

With one unit of CPU work a single processor can handle a throughput of approximately 94 Mbps. Adding more processors to the system decreases the throughput to slightly over 80 Mbps because the extra processes introduce overhead and the computational requirements demanded of the processes does not produce enough speedup to counteract the overhead.

As more slaves are utilized the master must pass more data back and forth between them, and in the meantime, the slaves are more likely to be idling instead of performing useful computations. These overheads will, in certain cases, overcome the extra processing capacity brought online and thus result in a lower throughput. As the number of units of CPU work

increases, however, the computational load required of the additional processors is high enough to overcome the overhead introduced, resulting in higher throughput rates.

Additionally, with one unit of CPU work, there is a very slight rise and fall as more processors are added because in a given node, even though there may be two CPUs in use, there is only a single network interface. For example, when moving from three to four processors, the throughput falls because both processors are contending for access to the single network interface. Since there is only one unit of CPU work being done, this contention causes a relatively large amount of overhead and the throughput falls slightly. When more units of CPU work are performed, this effect is drowned out by the additional computational work.

As the amount of CPU work performed increases, a corresponding rise in the factor of increase of throughput is seen because the extra processors are being utilized more effectively. This can be seen most effectively in Figure 3.3. As more work is performed and more processors are brought online, the idling overhead is reduced and the communication overhead is low enough that we see a significant increase in throughput. A theoretical factor of increase of two is expected when using two processors instead of one, but this is

unachievable in practice due to overhead. From Table 3.2, we see that while there is initially a decrease in throughput, as more work is performed, the factor rises toward to its theoretical limit. Similar results may be seen in the remainder of the table, though the maximum increase factor achieved tends to move further and further away from its theoretical limit.

3.3.2 Effects of I/O Activity

A similar set of experiments determining how throughput scales with varying I/O work was also performed, with the results given in Table 3.3 and Figure 3.4. Since we are only interested in I/O performance, there is no CPU work being done, although the associated function call overheads are still included. Table 3.4 and Figure 3.5 show the throughput increase in a more convenient fashion, so we can easily see the throughput improvement as more processors and disks are added. As in Section 3.3.1, each column of the table gives the increase factor over a single processor. It is important to note that each node in the cluster contains only a single hard disk, and so each pair of processors must share one hard disk.

An obvious observation for this case is that the throughput rates are substantially lower than in Section 3.3.1. Disk I/O operations introduce a larger

I/O Work Iter.	Number of Processors							
	1	2	3	4	5	6	7	8
1	2302.55	3084.43	4448.87	4675.77	8880.55	10632.24	12142.04	13121.60
2	1182.75	1566.49	2237.84	2748.58	3712.68	4375.16	5243.14	5546.79
4	698.25	794.51	1195.92	1403.65	1756.81	2097.34	2488.17	2599.78
8	376.75	401.36	623.34	698.88	893.90	1039.10	1219.22	1266.94
16	193.31	179.25	315.94	329.36	427.41	518.16	617.95	620.91
32	98.94	90.02	166.52	168.19	218.94	263.98	301.79	323.62
64	51.91	47.83	86.12	84.17	108.82	129.91	151.18	155.26

Table 3.3: Throughput rates in Kbps for I/O work

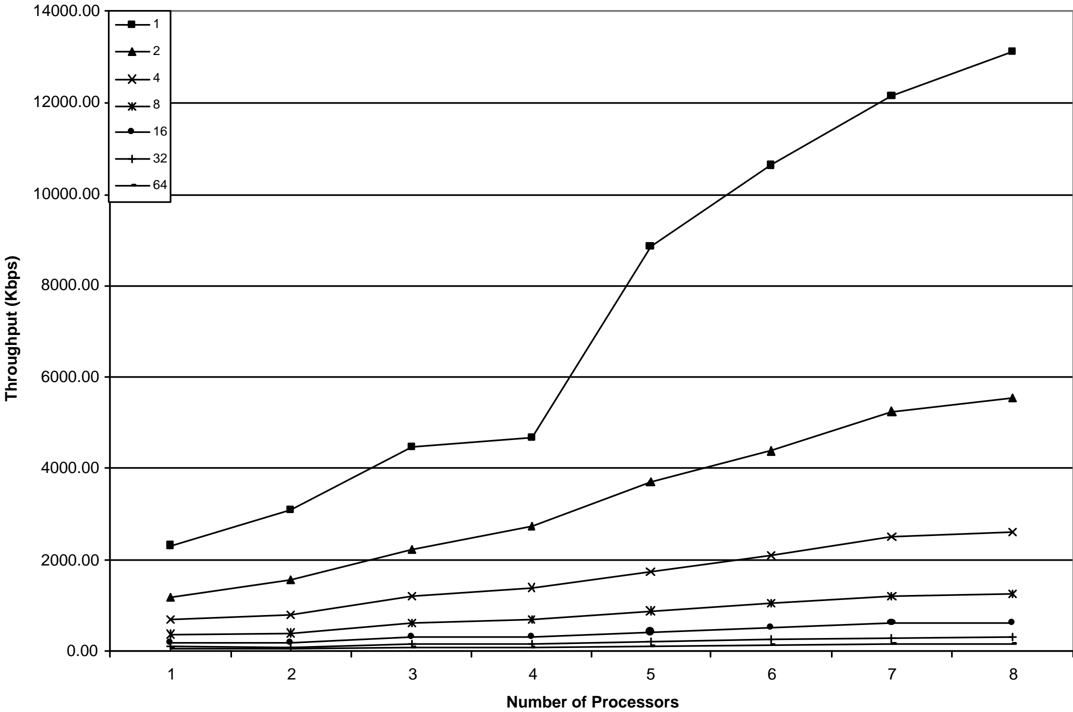


Figure 3.4: Throughput for I/O work

I/O Work Iterations	Throughput Increase Factor						
	2	3	4	5	6	7	8
1	1.34	1.93	2.03	3.86	4.62	5.27	5.70
2	1.32	1.89	2.32	3.14	3.70	4.43	4.69
4	1.14	1.71	2.01	2.52	3.00	3.56	3.72
8	1.07	1.65	1.86	2.37	2.76	3.24	3.36
16	0.93	1.63	1.70	2.21	2.68	3.20	3.21
32	0.91	1.68	1.70	2.21	2.67	3.05	3.27
64	0.92	1.66	1.62	2.10	2.50	2.91	2.99

Table 3.4: I/O work throughput increase factor over two processors

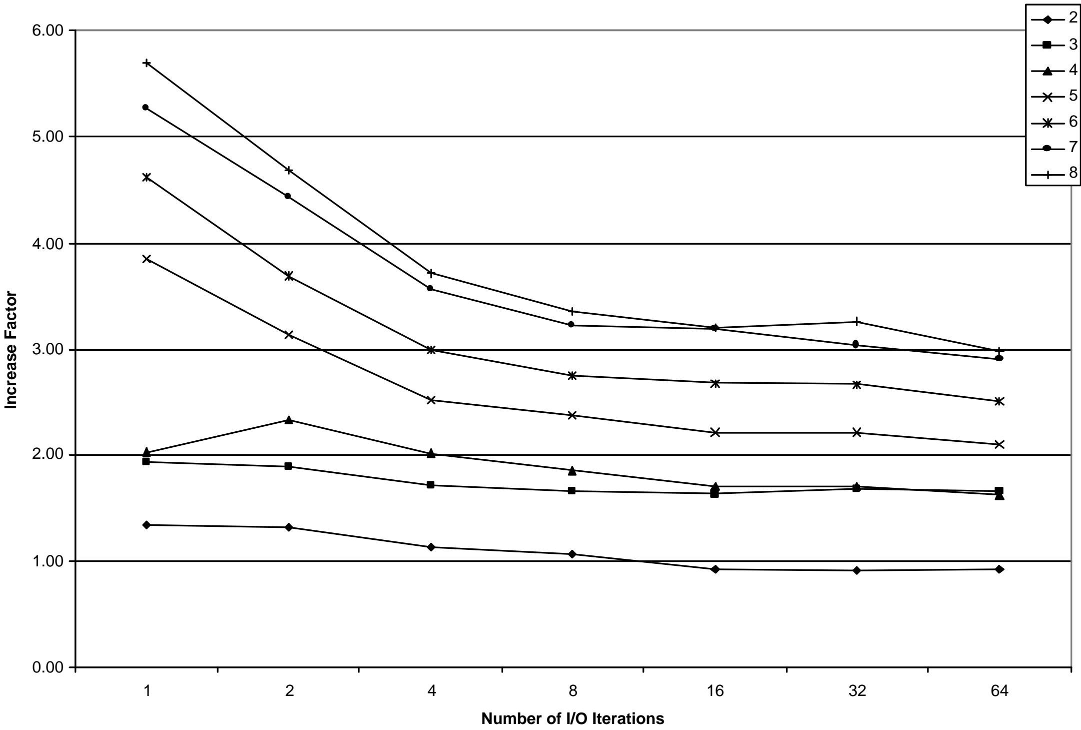


Figure 3.5: Throughput increase factor for I/O work

delay than CPU operations because they are on the order of milliseconds, whereas CPU operations are on the order of nanoseconds. Therefore, to obtain the best results in disk I/O, RAID is recommended for each node in the cluster to reduce the delay by automatically parallelizing disk operations.

Even though each pair of processors shares a single hard disk, using both processors of each node shows a clear improvement in throughput over using only a single processor. This improvement occurs because the disk is being utilized more fully due to the additional demands of the second processor. If a second disk were available in each node of the cluster, the throughput rates would no doubt be even higher.

With I/O performance, adding more processors and disks increases the throughput but the increase factor diminishes with more work as can be seen by the decline in the slope of Figure 3.5. That is, adding more processors and disks actually decreases the factor of increase of throughput as more I/O work is performed. From Table 3.4, we can see that in each column, as we increase the load to perform more iterations of I/O work, the throughput factor of increase over a single processor and disk is diminishing.

This performance decrease occurs due to an increase in idling overhead. When a slave is performing 64 iterations of I/O work, it takes a very large

amount of time, and in the meantime, the router continues to receive packets and attempts to pass them on to the master, while the master itself has a queue of outstanding packets that need to be sent to slaves for encoding. As a result, the router and master begin dropping packets because their buffers are overflowing. The master may also begin performing encoding operations itself if all slaves are occupied and it has some idle capacity. Further idleness overhead occurs when a slave finishes its I/O work and instead of immediately being sent more work by the master, must wait because the master is itself engaged in encoding. Periodically the master pauses the encoding step and sends more data to the slave. The time the slave spends idling is wasted.

3.3.3 Effects of Combined CPU and I/O Activity

Table 3.5 and Figure 3.6 show how the throughput (in Kbps) changes with the number of processors for combined CPU and I/O work units. This means that the stated number of iterations are performed for both CPU *and* I/O work. Table 3.6 and Figure 3.7 demonstrate the increase factor as the number of processors in use are scaled up. As in Sections 3.3.1 and 3.3.2, each column of the table gives the increase factor over a single processor.

When combining CPU and I/O work, we expect the I/O performance num-

Combined CPU & I/O Work Iter.	Number of Processors							
	1	2	3	4	5	6	7	8
1	2224.74	2920.43	4453.58	4706.93	7549.62	7025.05	8638.40	8529.30
2	1164.42	1515.37	2518.72	2802.96	3364.09	3646.64	4947.87	4680.94
4	686.76	735.37	1181.77	1357.73	1728.09	2091.76	2417.86	2424.72
8	376.01	384.41	622.29	681.40	886.73	1048.56	1200.13	1233.12
16	191.09	189.47	327.68	333.00	442.00	516.73	605.64	620.73
32	93.77	99.77	168.78	176.23	222.52	261.86	298.39	309.83
64	49.87	47.28	84.27	90.22	102.62	119.84	152.17	153.58

Table 3.5: Throughput rates in Kbps for combined CPU and I/O work

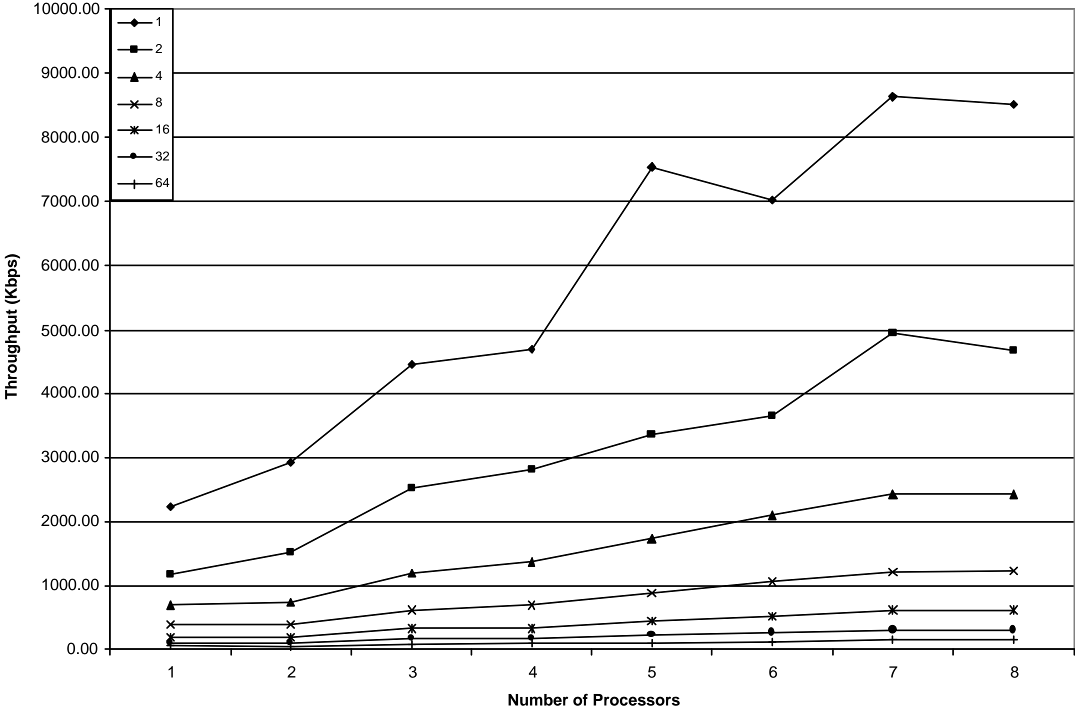


Figure 3.6: Throughput for combined CPU and I/O work

Combined CPU & I/O Work Iterations	Throughput Increase Factor						
	2	3	4	5	6	7	8
1	1.31	2.00	2.12	3.39	3.16	3.88	3.83
2	1.30	2.16	2.41	2.89	3.13	4.25	4.02
4	1.07	1.72	1.98	2.52	3.05	3.52	3.53
8	1.02	1.65	1.81	2.36	2.79	3.19	3.28
16	0.99	1.71	1.74	2.31	2.70	3.17	3.25
32	1.06	1.80	1.88	2.37	2.79	3.18	3.30
64	0.95	1.69	1.81	2.06	2.40	3.05	3.08

Table 3.6: Combined CPU and I/O work throughput increase factor over two processors

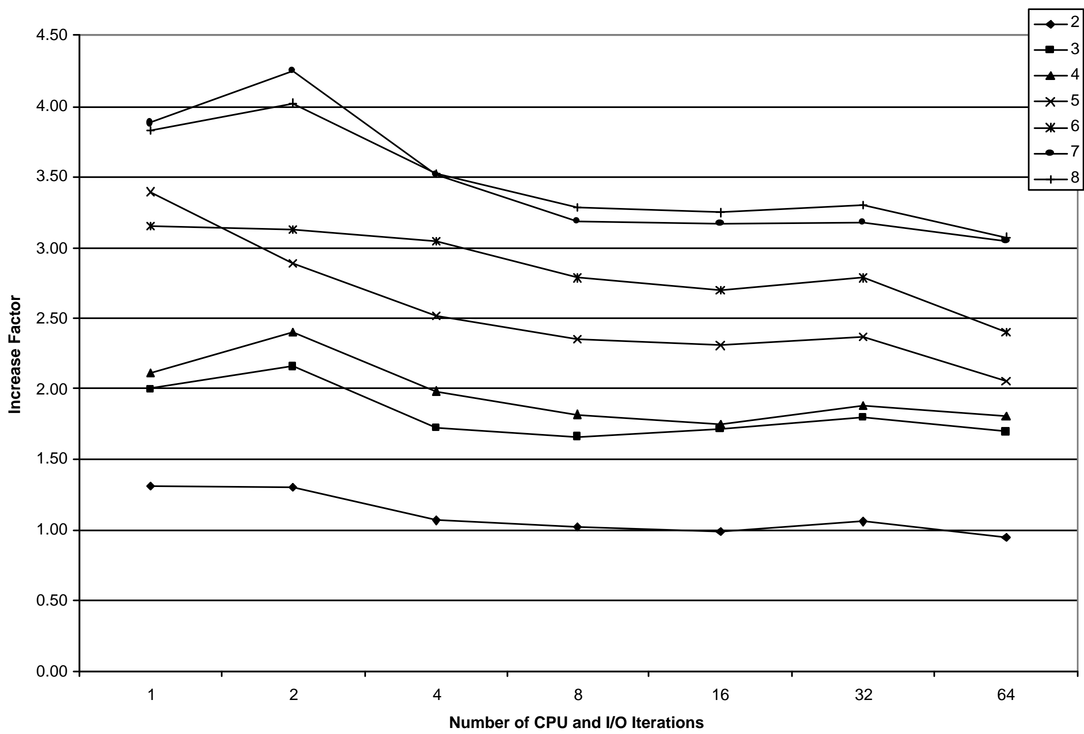


Figure 3.7: Throughput increase factor for combined CPU and I/O work

bers to dominate because they are significantly lower than the CPU performance numbers. A comparison of Tables 3.3 and 3.5 confirms this expectation. Because CPU work is also being performed, the numbers in this set of experiments tend to be slightly lower than the corresponding numbers in Section 3.3.2.

Since the I/O numbers dominate, we expect that adding more processors and disks will again reduce the throughput increase factor as more combined work is performed. Figure 3.7 confirms this but also shows that the decrease is shallower. While the I/O iteration increase is working to reduce the increase factor, the CPU iteration increase is trying to increase the same factor. As a result, the decrease is lessened. The analyses given in Sections 3.3.1 and 3.3.2 hold here as well.

3.4 Summary

This chapter presented a simulation environment that allowed us to verify the hypothesis that parallelizing the exhaustive monitoring system would allow us to scale to higher throughput rates. The simulation architecture and implementation were discussed and the results of several experiments were provided. The experiments confirmed that throughput scales well with an

increasing number of processors and disks for various combinations of CPU and disk I/O work. As we require more work to be performed per–packet, the factor of increase of throughput increases for CPU intensive tasks, but decreases for disk I/O–centric tasks. The next chapter discusses how packets are acquired from and then released back into the network, and also describes the concept of data streams.

Chapter 4

Acquiring and Encoding Packets

This chapter discusses the techniques used to transparently acquire packets from the network so that they may then be processed further. After being captured, the packets are separated into streams based on various factors. This separation benefits many encoding schemes and also aids with other types of categorization used in monitoring. If required, the packets are then encoded or decoded, and then released back into the network.

4.1 Packet Capture and Release

The most popular method of acquiring packets from the network is to use a C–language library called *libpcap* and to place the network interface into

promiscuous mode. The libpcap library is a framework for user–level packet capture [44] and it acquires packets at the data link layer using the underlying operating system's packet capturing facilities. It is mainly used for security monitoring and network debugging, which do not require packets to be returned back to the network, and as a result, it does not provide a means to do this.

Transparency is one of the requirements of the system and it was decided that this could be best accomplished by casting it in the form of a transparent Ethernet bridge using FreeBSD as the operating system. In addition to a bridge feature, FreeBSD provides a facility for applications to read and write Ethernet frames through a system device known as *tap*. The tap device is essentially a virtual Ethernet interface, and can be accessed by reading from and writing to the /dev/tap0 device.

This in itself is not particularly useful because the virtual interface is not connected to any networks and thus does not receive any network traffic. However, there is means of connecting a real Ethernet interface to the virtual one, and thus it is possible to read and write raw Ethernet frames on the network. Figure 4.1 illustrates how the tap virtual device works. Since this facility is provided directly by the operating system and supports both read-

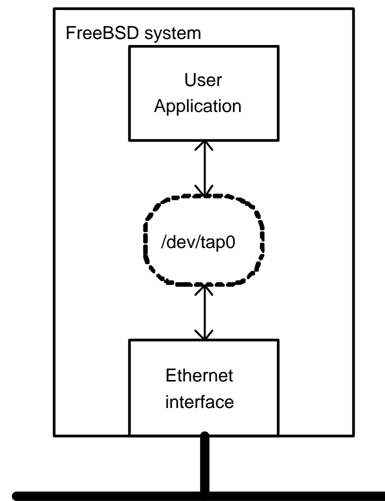


Figure 4.1: The tap virtual device

ing and writing, it was felt that this was a better choice for the system than libpcap. The tap kernel module can be loaded into the operating system from the command line, or compiled directly into the kernel. The latter option was chosen.

Using the tap device requires the system to manage Ethernet headers and trailers. Once a packet is obtained, the header and trailer must be saved before being stripped from the packet, since they are not needed during the encoding phase. Upon completion of encoding, it is necessary to add the header and trailer back. To save memory, the system stores only the MAC address and recreates the header and trailer when necessary. In order to match the MAC address with the correct packet, a record of the corresponding IP ad-

dress is stored and used to look up the MAC address.

4.2 Packet Encoding

With exhaustive network monitoring, all packets can be stored to disk for later analysis and/or classified in real–time. But performing exhaustive network monitoring also allows us to undertake packet transformation operations, something that is impossible with packet sampling. Transformations may include lossless or lossy compression to effectively create bandwidth, encryption to ensure security, and packet filtering to deny specific applications network access.

Many encoding schemes work as symmetrically paired operations, such as compression–decompression and encryption–decryption. This means a corresponding decoding operation is required before the packet is passed on to neighbouring networks or to the client and also that errors introduced during packet processing will render the data irretrievable on the other side.

It is not necessary that all packets be encoded. Each packet may be examined, and based on certain criteria, may be encoded or passed on unaltered. This examination of packets also permits a much more comprehensive means of filtering packets. Typical firewalls will allow or disallow packets based on

the values of certain fields within the IP, TCP, and/or UDP header. An exhaustive monitoring system we can do much more.

The payload of packets may be examined and filtered on the basis of the contents found and instead of merely granting or denying packets access, their contents can be modified before sending them off to their destinations. This modification may include encoding the data, adding information, removing sensitive information, or it exchanging payload data with other data.

There are many reasons why a network operator may wish to encode the data as it passes through their network. Lossless compression provides additional bandwidth, thus staving off the installation of expensive new infrastructure equipment. Compression also allows the network to handle high loads during peak times and creates bandwidth on rural or remote routes, making them more cost–effective. If sensitive data will be passing through an untrusted network, the packets may be encrypted in order to reduce the possibility of it being intercepted. It is also possible to employ a combination of packet transformations.

In the past, packet compression has not been used much due to the low compression rates achieved and the high overhead involved. The low compression ratios are mainly due to the fact that compression is performed

on a per–packet basis instead of taking advantage of correlated information across packets, as mentioned in Section 4.3, by combining related packets into streams, which significantly increases the achievable compression rate.

A stringent requirement of the system is that all encoding take place in real–time. The throughput of the network must not fall below the acceptable levels, and therefore the encoding schemes employed must be relatively fast and able to operate efficiently in streamed mode or on small blocks of packets that arrive together.

4.3 Maintaining State with Streams

As explained in Section 4.2, data compression is an effective method for increasing the throughput of a given network link because it reduces the amount of data that is sent. Most network data compression schemes use what is known as stateless compression, where each data packet is compressed independently. This precludes taking advantage of the correlation of the data over a series of packets and thereby prevents the system from achieving a satisfactory compression rate. Communication–specific compression algorithms, such as v.42 bis and v.44, compress each frame independently, which limits the redundancy that can be removed through compres-

sion, but are popular solutions due to their low complexity [45, 46].

Other data compression schemes, referred to here as single stream compression, do attempt to take advantage of the correlation of data across packets, however they consider all packets as belonging to a single stream. These schemes fail to recognize that the data being transmitted is a mixture from many different connections and there is little correlation between the connections. Each connection is sending and receiving packets that are related to one another within the connection, but packets between connections are not related. As a result, such schemes also fall short of the maximum potential compression ratio.

A superior scheme would attempt to separate those streams that are independent and at the same time try to maintain correlation across the packets. This scheme, while being far more complex than the above–mentioned schemes, produces the best compression rates. The saved state allows compression dictionaries to determine redundancies over all data in a connection, rather than just those existing within a single packet. Stateful compression introduces several difficulties, however. It requires that all packets be decoded in the correct sequence. Any dropped packets must be re–sent and the decompressor must wait for late packets to arrive before continuing.

The alternative is to resynchronize both sides by flushing their dictionaries, and though simpler, this technique would reduce the compression rate. In this thesis, it is assumed that a relatively error–free link is used, and that dropped or out–of–order packets are a rarity.

Streams are also useful when encrypting data. Each stream can be encoded with a different key, thus reducing the relationship between the packets and making it more difficult to break the encryption. According to Shannon [42] the more data an attacker receives that is encoded with the same key, the easier it is to break the code.

Stream ciphers and feedback–based block ciphers require the previous bits of ciphertext in order to produce the next bits of ciphertext. Therefore, maintaining state information is mandatory when using these encoding schemes.

4.3.1 Stream Separation Factors

Once we have determined that separating packets into streams will prove beneficial, the question of how to separate them arises. Finding the best solution to this problem is difficult, but worthwhile. A simple solution would be to create a stream for each unique set of network host and port pairs,

guaranteeing that different connections will be separated. This type of categorization is similar to that used in traffic flow monitoring. However, this scheme has several drawbacks. Firstly, even within a single connection there may be multiple data types being transferred. For example, in a single HTTP session there will likely be a mix of text and graphical data and combining them in a single stream will result in poor compression rates. Secondly, it would be beneficial to combine the same data types from all connections into a single stream in order to take advantage of the dictionary that has already been built from previous connection. For example, HTML data from all connections may be run through a single stream to maximize the compression rate. Most HTTP sessions are short–lived and could benefit greatly from this idea. Thirdly, if each possible combination of source and destination IP addresses and source and destination ports were placed in a separate stream, the memory requirements would be prohibitive if a large number of simultaneous connections were active in the network.

The monitoring system presented in this thesis provides a three–tiered solution, illustrated in Figure 4.2, to the problem of determining how to separate packets into streams so that those packets which have a high probability of correlation between them are passed through the same stream. At the first level, the source IP address is hashed into a bounded number of "major chan-

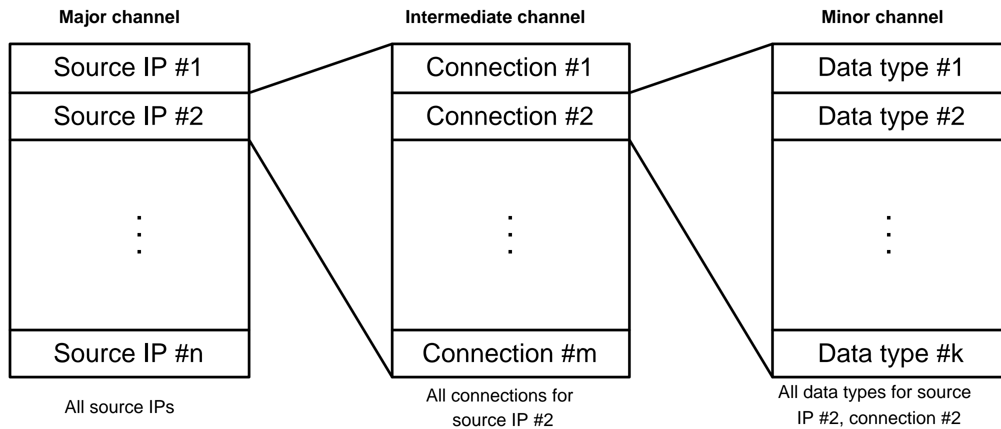


Figure 4.2: Stream separation scheme

nels". This separates connections that originate from different IP addresses because such data typically has the least amount of correlation. At the second level, the connections from a particular source IP address are separated into a fixed number of "intermediate channels," based on a hash of the source IP, source port number, destination IP, and destination port number. This separation reduces the intermixing of data between the many connections of a single host. Finally, the third level separates the data within a connection into a certain number of "minor channels" based on the type of the data in the connection. For example, HTML text might be one type, while images might be another. The number of major, intermediate, and minor channels is fixed so that the maximum memory requirements are known *a priori*.

To determine which data types should make up the minor channels, sev-

eral heuristics are used. For HTTP connections, the HTTP header is examined to determine the Multimedia Internet Mail Extensions (MIME) type information. This information indicates what data is being transferred in the connection. In other cases, the packet payload is examined in an attempt to determine what data is being carried by matching key strings for popular data types. If none of the key strings match, a final heuristic makes a best–guess effort at determining whether the data is ASCII text or not. Once a data type is determined for a connection, it is stored in a table and remains for the lifetime of the connection. This is necessary because it is usually only possible to determine the type of the data at the beginning of the data transfer. Only at that time is the MIME type or file header information provided and if it is not discovered then, it usually cannot be determined later.

4.3.2 Mapping Web Traffic into Streams

Web traffic typically transfers the same data from the web server to an end host and it would be beneficial to place this data into the same stream because the dictionary built up for a specific site may be reused by all hosts. This has the potential to allow for a much greater compression for short–lived connections which transmit only a small amount of data.

The exhaustive monitoring system presented here monitors HTTP requests and acquires the URLs contained therein. If a connection is being made to a web server, the steps discussed in Section 4.3.1 are bypassed and instead the URL is parsed, hashed, and mapped to a specific stream. The URL parser removes any query string information and distinguishes between absolute and relative URLs. In this fashion, all requests for a specific URL will be mapped to the same stream, while other traffic will continue to be mapped in the manner described in Section 4.3.1.

4.4 Summary

This chapter discussed techniques used to quietly capture raw Ethernet frames, allowing the system to appear transparent to the network. The scheme chosen was the tap interface provided by the FreeBSD operating system. Next, the uses and benefits of packet transformation operations, such as compression, encryption, and flexible filtering, were discussed. Finally, the concept of streams was presented. Streams provide many benefits and may be required for some encoding operations. Their employment also results in superior compression rates. An overview of how packets are separated into streams was given. The next chapter presents a complete implementation of the sys-

tem and determines how well throughput scales with an increasing number of processors and hard disks.

Chapter 5

System Model and Architecture

In Chapter 3, via simulator–based experiments we verified that throughput could be increased by employing additional processors and hard disks, but those experiments were run with a comparatively simple system on a parallel cluster. In this chapter, a complete exhaustive monitoring system is implemented on the same parallel cluster.

This system implements all of the features discussed in Chapter 4 and several more, introduced in this chapter. As such, it captures Ethernet frames at the data–link layer, uses heuristics to separate the packets into streams in order to increase the encoding function utility, encodes the packets, and then releases the frames back into the network. By functioning at the data–link

layer the system is transparent to the network, meaning that no network reconfiguration is required when the system is deployed. In addition, the system provides a resynchronization protocol, described later in this chapter, that provides recovery in the event of packet error or loss. The system presented in this chapter is quite large and far more complicated than the simulation and it is necessary to repeat the experiments run in Chapter 3 in order to determine if the throughput will scale in a similar manner.

5.1 System Architecture

The main goal of the system is to take advantage of packet encoding schemes while allowing for high throughput and maintaining transparency. To that end, the system has been designed as a transparent Ethernet bridge, meaning that the existing network is not aware that the system is there and does not need to be reconfigured in any way. Figure 5.1 shows how the transparent bridges fit into the network. The default deployment is a dual bridge system between the routers of a leased line or some other dedicated connection. This connection is assumed to be relatively error free, as described in Section 4.3, though a resynchronization protocol is used in the rare event of packet loss or corruption (see Section 5.3.2).

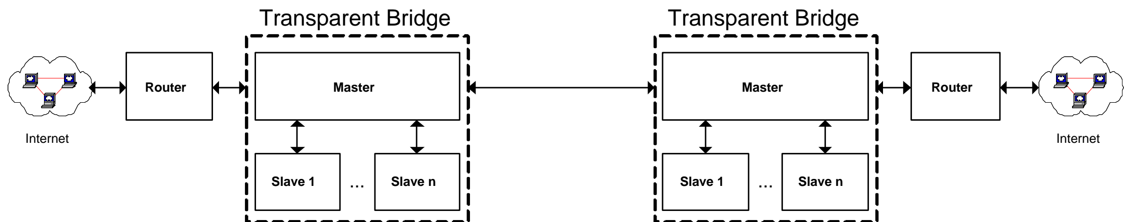


Figure 5.1: Transparent bridge architecture

Packets travelling through the link from one router to the next are passed through the bridge immediately after the router. Depending on the desired encoding, the packet may be transformed, and then decoded at the other end of the link. If so, the second bridge will take care of this operation before sending packets to the next hop router. The bridge operates at the data–link layer, but the packets may be filtered and transformed based on packet information at levels up to and including the application layer.

The bridges do not know which interface will receive packets from the router and which will receive encoded packets from the corresponding bridge. All received IP packets that bear a specific protocol number (e.g., in our case, 63) will be decoded, and all packets with other protocol numbers will be encoded and the protocol number will be changed to 63. The Internet Assigned Numbers Authority (IANA) [3] has assigned protocol number 63 for any local network, ensuring that it should not otherwise be seen on the link. This change in protocol numbers will only take place if the packet is transformed

and needs to be decoded at the other end of the link.

The entire system is built using standard off–the–shelf components and FreeBSD's bridge and tap services. The bridge machines are capable of parallel processing via a parallel cluster and if necessary, contain several hard disks.

5.2 Load Balancing

A slave process is created and assigned to each CPU available. This process encodes or decodes the data and also maintains state for the data streams it has been assigned. The master process receives all incoming packets from the tap interface, determines which operation to perform, and then sends the packet to the appropriate slave processor. When the data is returned, the master process sends the packet out of the second tap interface, completing the bridge function. All interprocess communication takes place via sockets.

The algorithm used to map packets to a process is very simple. The source and destination IP address of the packet are hashed and taken modulo n, where n is the number of processors. This ensures that all packets in a connection are assigned to the same stream on the same processor so that the system takes advantage of inter–packet correlation. Assuming a well

designed hash function, this scheme should distribute the packets roughly evenly between all processors. The hash function used is an exclusive–OR operation of the two addresses.

Using the TCP or UDP port numbers in the hash would be beneficial but this information is not always available because it may be the case that the encoding operation does not leave a TCP or UDP header within the IP packet. For example, in the implemented system the encoding operation is compression and several small packets are compressed into a larger "package" and placed inside an IP packet (see Section 5.3.1 for further details). The package does not contain a TCP header and therefore does not have a port number.

5.3 Implementation

The exhaustive monitoring system has been implemented with the example encoding function being data compression and with the goal of maximizing the compression ratio. The zlib library was chosen for compression because it offers good compression rates and compresses streamed data effectively. zlib is a free, general–purpose lossless data–compression library that provides in–memory compression and decompression functions [26]. Switching to other encoding methods would not be difficult.

As with the simulation, the computations are abstracted into units of CPU and disk I/O work. One unit of CPU work is defined as the time required for compressing an incoming packet using zlib and also performing header compression (see Section 5.3.1). One unit of I/O work is defined as opening a new file handle, writing the uncompressed packet to the local disk, closing the file handle, re–opening it, and then reading back the file back. All disk writes are followed by a call to the *fsync* library function, which forces data to be written to the disk instead of stored in cache memory for later writing.

The complete system was implemented on the same computers as used in the simulation, dual–processor Pentium III 500 MHz machines with 512 MB RAM running FreeBSD. All components were connected via 100 Mbps Fast Ethernet. The code was written in C++ and is approximately 8900 lines.

5.3.1 Combining Packets into Packages

Many applications, such as telnet sessions or short database transactions, send large numbers of small packets. These packets often contain only a few bytes of data, whereas the header information is often in excess of 40 bytes (20 bytes for the IP header and at least 20 bytes for the TCP header). Compressing such a small packet would not produce much benefit, particularly

when coupled with the fixed 6–byte overhead introduced by zlib.

To overcome this problem, multiple packets within a connection are packaged together into a single, larger packet, or “package”, allowing compression to occur over a larger amount of data. This permits the compressor to find more redundancy across packets and the 6–byte zlib overhead is only paid once per package. The downside is that the header information must be stored within the package payload because there are multiple packet headers contained within the same package.

Since there is usually no correlation between the packet header and payload, compressing them together would not produce the best results. A better solution would be to separate the header from the payload and use a separate compressor for each part. We continue to use zlib for the payload, but the header is compressed with a system similar to that presented by Jacobsen [32].

Our system implements a modified and simplified Jacobsen header compression scheme that results in a slightly worse compression due to the simplifications introduced. Like Jacobsen's, our scheme is specific to TCP/IP datagrams but instead of producing an average compressed header size of 3 bytes, our scheme averages 4 bytes but is much simpler to implement. The

compressed headers within a package are then further compressed with zlib.

When the transparent bridge receives a frame from the tap interface, it removes the Ethernet header and trailer in order to process the IP packet. These must be restored when the packet is returned to the network, so the MAC address and corresponding IP address are saved and the Ethernet header and trailer are re–created when the packet exits the bridge. Because the package must be wrapped in an Ethernet frame, only those packets that have the same source and destination IP address as well as being compressed in the same stream may be packaged together. This limitation on which packets may be packaged together causes the average package size to be smaller, reducing the potential efficiency gains.

Even though it is ensured that packages are no larger than the maximum IP packet size, the compression transformation may create packages that are larger than the Maximum Transmission Unit (MTU) size, a data–link layer restriction. To avoid this, it may be necessary to perform packet fragmentation and reassembly. This task that has been implemented in and is performed by the monitoring system software because this capability is not provided by FreeBSD's bridge feature.

5.3.2 Resynchronization

If packet loss occurs between the bridges it is imperative that the system somehow detect it and take steps to correct it because otherwise the encoding stream state would not match between bridges and decoding would not be possible. To serve this purpose, a simple, low overhead resynchronization protocol has been developed.

The identification field of the IP header is used as a sequence number for packages in order to recognize when packets have been lost. The original IP headers have been compressed and bundled into a package, so no information is lost. The sequence number runs between the values 1 and 4095, incrementing by one for each package. The value 0 is reserved for initial startup and resynchronization.

When an unexpected sequence number is received, a resynchronization action is undertaken. The system maintains a small out–of–order queue which buffers packets that have arrived before they were expected (i.e. the sequence number of the received packet is higher than the next expected sequence number). If the sequence number of the received packet is larger than the expected sequence number by an amount no greater than the out–of–order queue size, it is stored in the queue, unless the queue is full. For

example, suppose the out–of–order queue can store only one packet, and the next expected sequence number is 1000. If the packet that arrives next has a sequence number of 1001, the packet will be stored in the queue. If, however, the packet has a sequence number greater than 1001, it will be discarded because the queue space is not large enough to store more than one packet ahead. Received packets with a smaller than expected sequence number are also dropped because they are assumed to be duplicates. Once the queue is full, resynchronization must commence and all packets in the out–of–order queue are dropped. They are no longer needed because the system will re–send all packets beginning with the next expected packet.

The queue size is currently set at a value of four packets to ensure that dropped packets are quickly recognized and resynchronization is not excessively delayed. The network link is assumed to be virtually error–free and so this protocol should be executed only rarely.

When an expected packet is received and the queue is not empty, the packet is first processed, the next expected sequence number is incremented, and the out–of–order queue is scanned to determine whether any of its packets follow the current one. If so, that packet is then processed and the queue is again examined for further packets in the sequence.

When resynchronization is determined to be necessary, the bridge receiving the data sends a resynchronization request back to the sending bridge. After the request is made, the receiver awaits confirmation. Upon receiving a resynchronize request or request confirmation, all state information is cleared, and the sequence number is reset to 0. When this happens to the system implemented here, the compression rate drops because all compression–related state information must be completely rebuilt. This should be a rare event though, because of the mostly error–free link. If any package is received while waiting for a confirmation, the packet is dropped and another reset request sent. It is left to higher layers such as TCP to perform recovery of all dropped packets.

5.4 Performance

Experiments were run to determine how well the system encodes data, and also to determine how well it scales with additional processors. Several gigabytes of popular websites such as yahoo.com, cnn.com, and microsoft.com were saved to a local web server in order to reduce the length and variability of network latency. Local hosts retrieved the web pages using multiple web crawlers over a link containing the compressing Ethernet bridges. To take

Compression Scheme	Rate
Per–Packet	1.70
Single Stream	2.19
Monitoring and encoding system	2.79

Table 5.1: Achievable compression rates

advantage of the URL mapping discussed in Section 4.3.2, the UNIX hosts file used for hostname to IP address resolution was modified so that the external domain names pointed to the local server. As a result, the bridge sees the correct URL request. Depending on the particular goal of the experiment, certain amounts of CPU and disk I/O work were performed on the packets.

5.4.1 Compression Rates

The host ran eight web crawlers with each experimental run requesting URLs from the web server in the same random order. That is, the order of the requests was generated randomly, but the same order was used in each experiment to prevent a change in the compression rate due to the request order. Table 5.1 summarizes the results of the experiments.

From the table, we can see that compressing each IP packet individually and then discarding the compression dictionary afterward results in the compressed data being 1.70 times smaller, the poorest showing. Maintaining a

single stream of state across packets gives a better compression rate, but neither method is nearly as good as the complete system described in this chapter.

5.4.2 Effects of CPU Activity

These experiments determine the maximum throughput of the system with varying amounts of CPU work. Table 5.2 and Figure 5.2 illustrate how the throughput (in Mbps) changes with the number of processors. There is no I/O work being done in these experiments. Table 5.3 and Figure 5.3 show the factor of increase in throughput from using additional processors. The Ethernet bridge requires the use of two machines in the cluster, so there remain only six to use for experiments.

Tables 5.2 and 5.3 include data for zero CPU work units in order to obtain an upper bound on the system throughput. In this situation, adding extra processors leads to small gains because we are introducing communication overhead but are not taking advantage of the extra processors by performing any useful operation. Unlike the simulation however, a throughput increase is seen in this case because this system performs more work on each packet, such as stream separation heuristics, even when the packet is not

CPU Work Iterations	Number of Processors					
	1	2	3	4	5	6
0	56.80	53.92	57.68	60.96	62.64	64.32
1	18.96	18.72	23.52	28.40	31.60	36.88
2	16.72	16.96	21.84	26.32	30.08	33.36
4	13.44	16.56	19.04	23.04	26.16	30.72
8	9.52	14.64	15.44	17.92	21.36	25.68
16	4.03	8.32	10.48	13.12	15.44	18.16
32	3.69	5.23	6.13	8.16	10.64	11.84
64	1.94	2.92	3.16	4.06	5.46	5.82

Table 5.2: Throughput rates in Mbps for CPU work

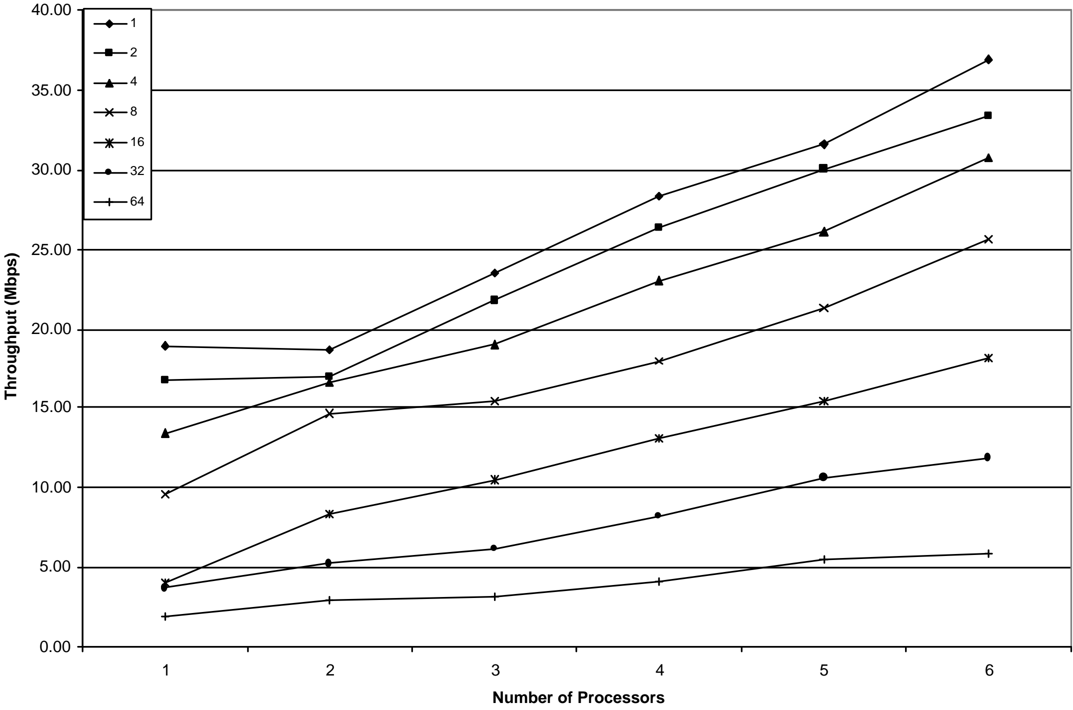


Figure 5.2: Throughput for CPU work

CPU Work Iterations	Throughput Increase Factor				
	2	3	4	5	6
0	0.95	1.02	1.07	1.10	1.13
1	0.99	1.24	1.50	1.67	1.95
2	1.01	1.31	1.57	1.80	2.00
4	1.23	1.42	1.71	1.95	2.29
8	1.54	1.62	1.88	2.24	2.70
16	2.06	2.60	3.25	3.83	4.51
32	1.42	1.66	2.21	2.88	3.21
64	1.51	1.63	2.09	2.81	3.00

Table 5.3: CPU work throughput increase factor over one processor

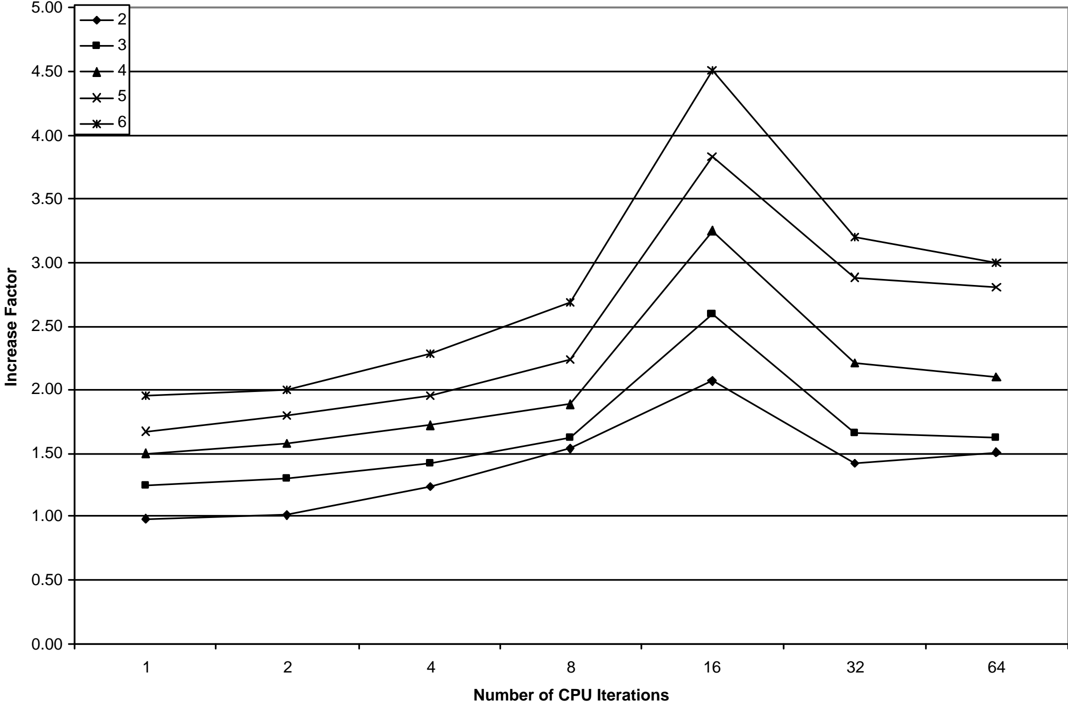


Figure 5.3: Throughput increase factor for CPU work

compressed.

When performing only one unit of CPU work, the throughput rates increase slowly, until using six processors yields slightly less than double the throughput of a single processor. Increasing the iterations of CPU work continues to realize gains, though they are small at first. As more work is performed per CPU, the overhead becomes smaller in comparison to the computational gains, and so the throughput increases very nicely as 64 units of work are performed. Figure 5.3 shows that the system reaches a maximum throughput increase factor at 16 iterations of CPU work, but this does not mean that the throughput decreases after this point, as Table 5.2 and Figure 5.2 clearly show. It means that performing 16 units of CPU work provides the highest growth rate in throughput as more processors are added.

The absolute throughput rate in the system is lower than that of the simulation because there is much more overhead for each packet. In the simulation, the master simply passed all packets to the slaves and did not perform any other tasks that would be necessary in a complete system. In the complete system the Ethernet header must be removed and the re–created later, the packet must be split up into streams, the headers are compressed twice using different compression methods, and the resynchronization proto-

col may need to execute.

5.4.3 Effects of I/O Activity

This set of experiments determines how well I/O work scales. The results are given in Table 5.4 and Figure 5.4. There is no CPU work being done, although the overhead for associated function calls is still included. Table 5.5 and Figure 5.5 show the throughput increase factor in a more convenient manner. Each column of the table gives the increase factor over one process. As noted before, each node in the cluster contains only a single disk, meaning that each pair of processors share one hard disk.

Once again, the throughput rates are substantially lower than those of Section 5.4.2, for the same reasons as with the simulation. We again see that though each pair of processors shares a single hard disk, using both processors of each node shows an improvement in throughput over using only a single processor in the node. This improvement is seen because the disk is being more fully used due to the additional disk requests of the second processor.

Adding more processes and disks increases the throughput as expected, but when we reach 64 iterations of I/O work, using only one or two proces-

I/O Work Iterations	Number of Processors					
	1	2	3	4	5	6
1	571.92	794.97	1109.52	1178.16	2104.67	2579.36
2	273.44	366.41	497.66	609.77	883.21	1008.99
4	137.12	160.43	242.70	281.10	351.03	462.09
8	64.40	66.33	103.68	119.14	145.54	180.32
16	38.16	37.02	62.58	66.02	80.52	93.87
32	20.08	19.68	32.73	34.14	45.98	56.63
64	—	—	18.81	18.39	23.77	25.74

Table 5.4: Throughput rates in Kbps for I/O work

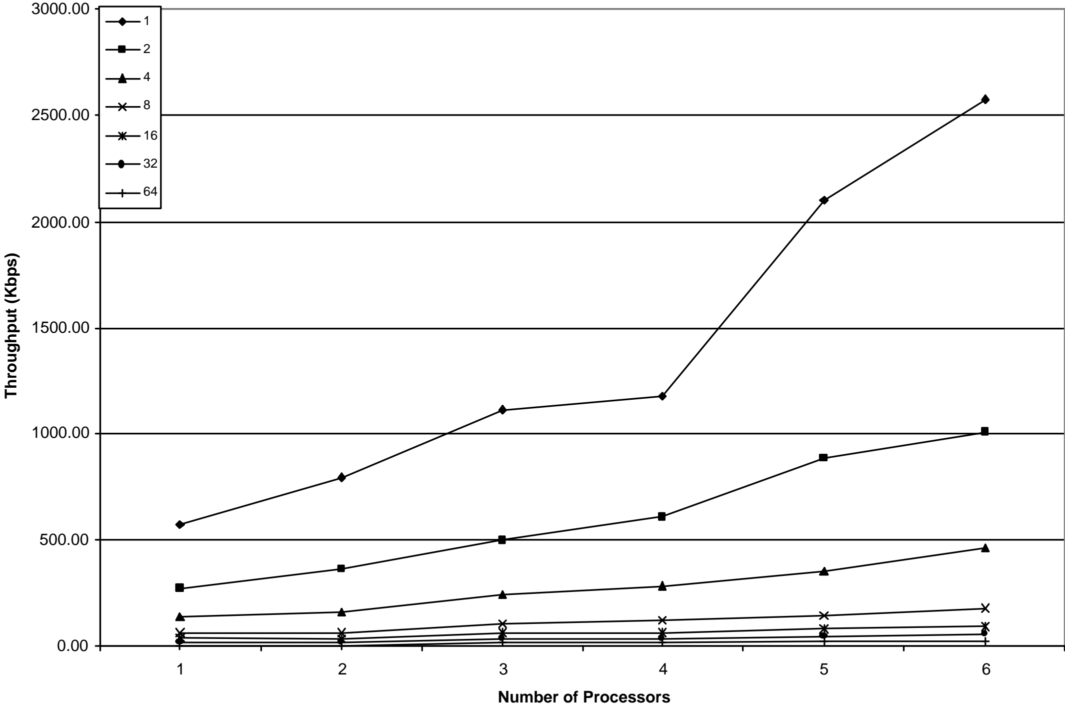


Figure 5.4: Throughput for I/O work

I/O Work Iterations	Throughput Increase Factor				
	2	3	4	5	6
1	1.39	1.94	2.06	3.68	4.51
2	1.34	1.82	2.23	3.23	3.69
4	1.17	1.77	2.05	2.56	3.37
8	1.03	1.61	1.85	2.26	2.80
16	0.97	1.64	1.73	2.11	2.46
32	0.98	1.63	1.70	2.29	2.82

Table 5.5: I/O work throughput increase factor over one processor

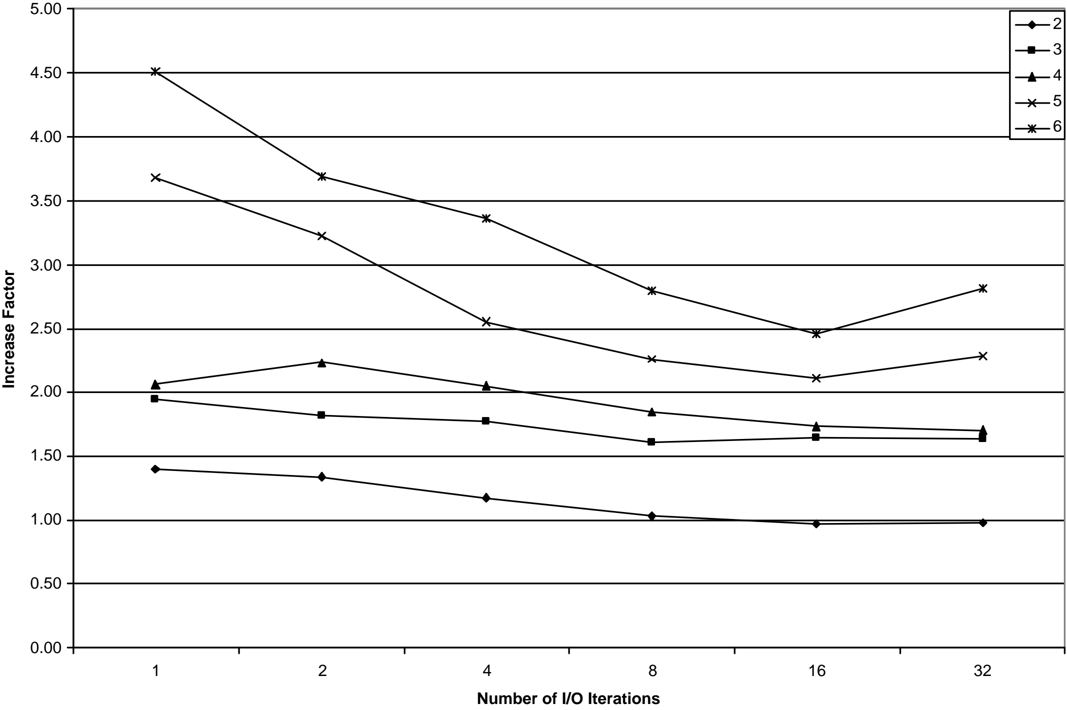


Figure 5.5: Throughput increase factor for I/O work

sors, the experiment does not complete because the timeouts grow too large for TCP to work reliably. However, with three processors, the test with 64 iterations is able to complete. As with the simulation, there is a decrease in the throughput increase factor as more I/O work is performed.

5.4.4 Effects of Combined CPU and I/O Activity

Table 5.6 and Figure 5.6 give the results for experiments performed to determine how the throughput (in Kbps) changes with the number of processors for combined CPU and I/O work units. The same number of iterations for both CPU *and* I/O work are performed. Table 5.7 and Figure 5.7 demonstrate the increase factor as additional processors are used.

As with the simulation, the I/O performance numbers dominate because they are significantly lower than the CPU numbers, as can be seen by comparing Tables 5.2 and 5.4. Overall, the throughput in these experiments is comparable with those of Section 5.4.3 despite the additional CPU work being performed. Once again, experiments with 64 iterations using one and two processors would not complete due to TCP timeouts, but adding additional processors increased the throughput to the point that the test could complete.

Combined CPU & I/O Work Iterations	Number of Processors					
	1	2	3	4	5	6
1	603.36	826.60	1315.32	1357.56	2015.22	2033.32
2	330.48	442.84	717.14	816.29	958.39	1060.84
4	164.56	184.31	281.40	327.47	421.27	495.33
8	69.04	71.11	117.37	122.89	178.12	187.79
16	35.92	35.56	63.94	62.50	83.69	96.98
32	19.96	20.56	36.53	36.53	47.70	56.49
32	—	—	18.81	19.34	23.45	28.57

Table 5.6: Throughput rates in Kbps for combined CPU and I/O work

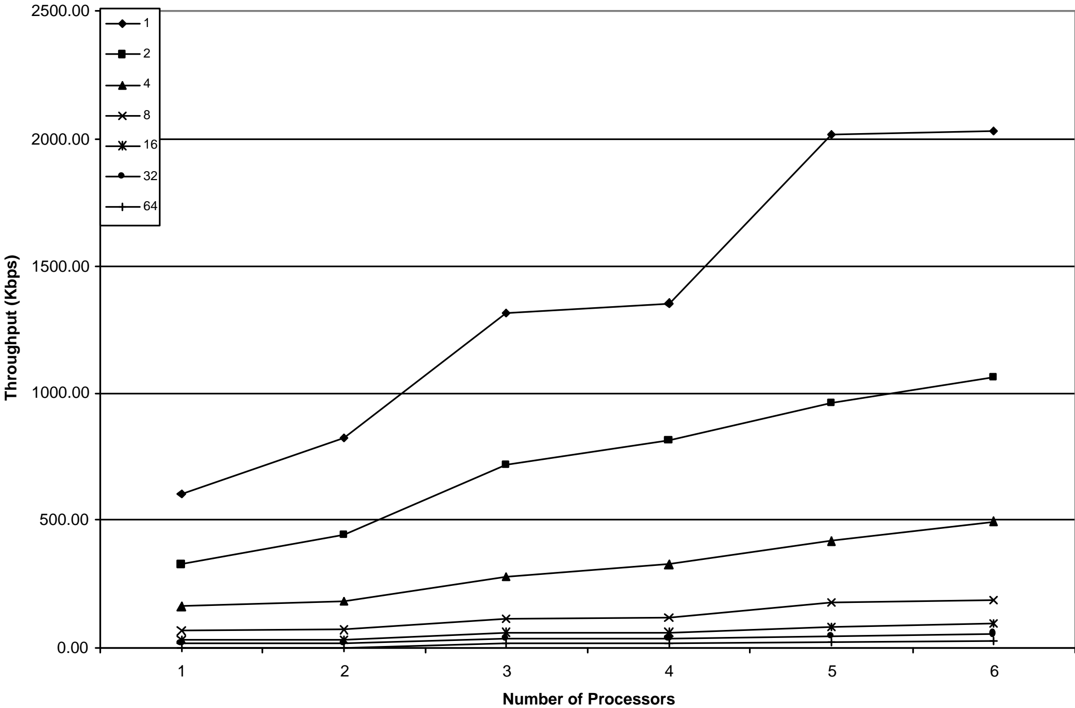


Figure 5.6: Throughput for combined CPU and I/O work

Combined CPU & I/O Work Iterations	Throughput Increase Factor				
	2	3	4	5	6
1	1.37	2.18	2.25	3.34	3.37
2	1.34	2.17	2.47	2.90	3.21
4	1.12	1.71	1.99	2.56	3.01
8	1.03	1.70	1.78	2.58	2.72
16	0.99	1.78	1.74	2.33	2.70
32	1.03	1.83	1.83	2.39	2.83

Table 5.7: Combined CPU and I/O work throughput increase factor over one processor

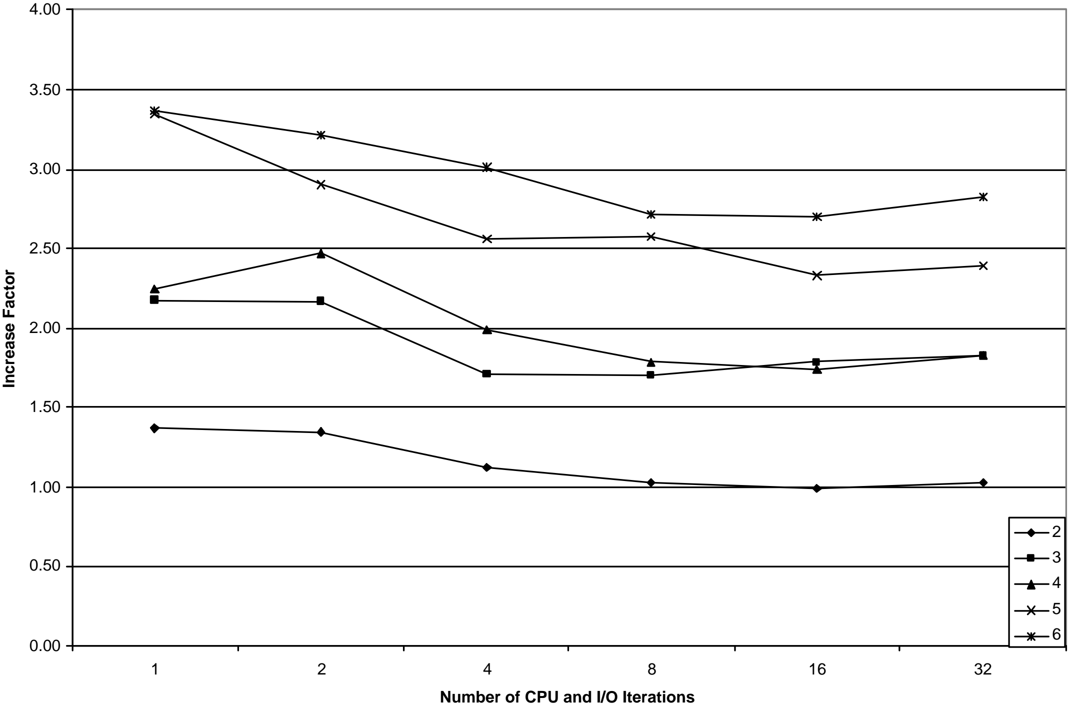


Figure 5.7: Throughput increase factor for combined CPU and I/O work

As with the simulation, adding more processors and disks results in an decrease in the throughput increase factor as more combined work is performed. The rising CPU work increase factor is not enough to overcome the falling I/O work increase factor, resulting in the overall decrease seen in Figure 5.7.

5.5 Summary

This chapter presented a complete system for exhaustive monitoring and encoding on a parallel cluster. The system contained several optimizations designed to improve compression rates, such as header compression and combining several small packets into a larger package. It also provided a resynchronization protocol in the event of packet loss. Experiments were run to determine how well the system scaled when more processors were added. As expected, based on the simulation results, very good improvements were achieved in terms of scaling, particularly as the amount of work performed per–packet increased, though unlike the simulation, the system experienced a peak throughput increase factor at 16 iterations of CPU work. The next chapter provides ideas for future research directions and draws conclusions from this research.

Chapter 6

Future Work and Conclusions

The thesis investigated the feasibility of exhaustive network monitoring and encoding of data packets in real–time with the aid of a parallel processor and load balancing software. This chapter provides a critical evaluation of the system, discusses some avenues of investigation to further improve on the achieved system performance, and then draws conclusions from the research.

6.1 Critical Assessment

This section evaluates the system presented in the thesis in the following areas: exhaustive monitoring and encoding, transparency, and scalability.

6.1.1 Exhaustive Monitoring and Encoding

Most network monitoring systems in use today rely on some form of packet sampling because exhaustive monitoring is considered too difficult to achieve. As discussed in Chapter 2, packet sampling has several problems, the main one being that it produces less than completely accurate results. This thesis proposed the use of parallelism to capture every packet, so that the monitoring system would be perfectly accurate.

In addition to providing complete monitoring, an exhaustive system would be able to perform encoding operations on the data packets. For example, through data compression, the network could create bandwidth during peak loads and through encryption, the network could provide additional security.

Using only off–the–shelf hardware and an open–source operating system, a system capable of capturing every packet and encoding it online, in real–time was successfully created. The system also features a resynchronization protocol to recover from packet loss or error.

An example encoding function of data compression was used to verify the encoding capabilities of the system. By maintaining state in the form of data streams and through the use of intelligent heuristics, the system was able to achieve a compression rate of 2.8, versus only 1.7 with no state information

saved, an improvement of 65%.

6.1.2 Transparency

Telcos and ISPs must keep their networks running at all times, and taking a section of it offline to install the monitoring system is undesirable. Furthermore, if the monitoring system requires changes to the network configuration, either in hardware or in software, the network downtime is increased.

The exhaustive monitoring system presented in this thesis can be inserted into the existing network without any configuration because it has been modeled as a transparent Ethernet bridge. The only modification to the network that needs to be made by the network operators when it is deployed is the physical wiring that connects the system to the network. The bridge has no IP address of its own and is invisible to the network.

6.1.3 Scalability

A key objective of the monitoring system is scalability. It is expected that greater amounts of work can be performed per–packet, and higher throughput rates can be achieved by adding a larger number of processors to the system. No software changes should be necessary.

The monitoring system uses a single master process to send encoding work to slaves for completion. There is no limit to the number of slaves that can be active. From Chapter 5, when performing 16 units of CPU work, adding five more processors increased the throughput by 4.5 times over that of a single processor. Based on the simulation results of Chapter 3, there is good reason to believe that the trend of increased throughput with more processors and hard disks would continue.

6.2 Future Work

While fully functional, the system presented in the thesis is still a prototype. There are several research directions that can be investigated with the aim of increasing the maximum achievable system throughput, and this section will outline some of them.

6.2.1 I/O Performance

RAID would likely speed up I/O considerably and is recommended for systems that require high I/O performance. Fraleigh et al. [25] used a 5–drive RAID to dump packet traces and were able to keep up with OC–48 (2.5 Gbps).

Some encoding schemes require disk I/O and further examination of parallel I/O is desirable.

6.2.2 Load Balancing

The load balancing software assigns an incoming packet to a processor based on the source and destination IP addresses. Assuming an even distribution of addresses this means that the processors are approximately evenly loaded, but this is not always a valid assumption. There are a handful of web sites on the Internet that dominate all web traffic and a small percentage of users that tend to use a disproportionate amount of bandwidth. This might result in poor load balancing, diminishing the benefits of parallelization. It would, therefore, be useful to keep track of the load assigned to each processor, as is done with the simulation software. When a processor's load exceeds a given threshold, some of its work can be assigned to other processors instead.

This task is far more complicated in the exhaustive monitoring system because of the use of data streams. When work originally destined for a specific slave is instead sent to another slave, that new slave will have no state information corresponding to the pre–existing connection and as a result must recreate the state. All state information that was gained by the

original slave is lost. In addition, the sending side must somehow indicate to the receiving side that the stream information has been reset, so that the receiving side slave will also reset its information, and the states on the two sides will match.

It is possible to migrate the stream from the original slave to the new slave's memory space, and when the load on the original slave is lessened, to re–migrate the data back, but this is very complicated, and the migration process will increase the load on the already constrained memory bandwidth.

Each slave process maintains a fixed number of streams, and each connection is hashed to a particular stream, as explained in Section 4.3.1. Therefore, when a stream is migrated from one process to another, there may be a collision, resulting in a clash of streams. For the encoding function of compression, this would reduce the achievable rate of compression since two separate connections, with possibly little or no correlation between them, would be sharing a single stream. For the encoding function of encryption, this clash could be disastrous because if the encryption scheme is based on a feedback model, such as output feedback, the previous state information will be corrupted, and the output of the encryption function will not be correct.

6.2.3 Other Areas

The system uses sockets to communicate between processes, even those processors that reside within the same machine. For the case of multiple processors in the same machine, there are other interprocess communications methods that may be worth examining, particularly shared memory, which is generally believed to be the fastest [28]. The downside to shared memory is that it requires additional process synchronization, which tends to make the software very complicated and is difficult to do, so it is uncertain if the performance would increase.

Finally, since an open–source operating system is being used, it may be possible to optimize it specifically for the exhaustive monitoring system. This might include increasing the size of network buffers and other tweaks to heavily used subsystems.

6.3 Conclusions

Online and real–time exhaustive data monitoring and encoding is a challenging proposition. In the past, researchers have opted for monitoring based on sampling. This thesis has presented an architecture and system for exhaus-

tive data traffic monitoring on a parallel cluster that uses only off–the–shelf hardware, an open–source operating system, and custom software. Moreover, the system can be used not only for exhaustively monitoring and classification of network traffic, but also for encoding and filtering operations such as lossless data compression and encryption. Furthermore, the system is completely transparent to the network and requires no configuration.

With typical web traffic achieving a compression ratio of approximately 2.8, telecommunications companies and ISPs can effectively increase their bandwidth, or perform exhaustive monitoring. Network operators can seamlessly ensure the security of their data if it must pass through untrusted networks before reaching the trusted destination.

The experiments conducted show that a significant amount of online data packet processing can be performed and that the throughput increases when more processors are added to the task. As more CPU work is performed per–packet, adding more processors provides an even greater increase factor in throughput. When performing 2 units of CPU work on data, moving from one to six processors provides a doubling of throughput, but with 16 units of work, moving from one to six processors provides a throughput increase of 4.5 times. These results demonstrate the scalable nature of the system.

Glossary

FPGA Field–Programmable Gate Array.

IANA Internet Assigned Numbers Authority.

ICMP Internet Control Message Protocol.

IETF Internet Engineering Task Force.

IPC Interprocess Communication.

IPMP IP Measurement Protocol.

ISP Internet Service Provider.

MAC address Media Access Control address.

MIMD Multiple Instruction, Multiple Data.

MIME Multimedia Internet Mail Extension.

MTU Maximum Transmission Unit.

NTP Network Time Protocol.

PE Processing Element.

psamp IETF packet sampling working group.

RAID Redundant Array of Inexpensive Disks.

RTFM IETF Real–time Flow Measurement working group.

SIMD Single Instruction, Multiple Data.

SMP Symmetric Multiprocessor.

SNMP Simple Network Management Protocol.

SONET Synchronous Optical Network.

SPMD Single Program, Multiple Data.

Telco Telecommunications company.

Bibliography

[1] IETF packet sampling (psamp) working group. http://psamp.ccrle.nec.de.

[2] IETF realtime traffic flow measurement (rtfm) working group. http://www.ietf.org/html.charters/OLD/rtfm-charter.html.

[3] Internet Assigned Numbers Authority. Protocol numbers. http://www.iana.org/assignments/protocol-numbers.

[4] Jean Bacon and Tim Harris. *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003.

[5] Nevil Brownlee, Cyndi Mills, and Greg Ruth. Traffic flow measurement: Architecture. RFC 2722, October 1999.

[6] Jeffrey D. Case, Mark Fedor, Martin Lee Schoffstall, and James R. Davin. A Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[7] Guang Cheng and Jian Gong. Traffic behavior analysis with Poisson sampling on high-speed network. In *Proceedings of International Conferences on Info–tech and Info–net 2001 (ICII'2001)*, pages 158–163, 2001.

[8] Kenjiro Cho, Ryo Kaizaki, and Akira Kato. An aggregation technique for traffic monitoring. In *Proceedings of the 2002 Symposium on Applications and the the Internet (SAINT'02)*, 2002.

[9] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive random sampling for load change detection. Technical Report TR-01-041, University of Minnesota, 2001.

[10] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive packet sampling for flow volume measurement. Technical Report TR-02-040, University of Minnesota, 2002.

[11] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive random sampling for traffic load measurement. In *Proceedings of IEEE Interna-*

tional Conference on Communications 2003 (ICC '03), volume 3, pages 1552–1556, May 2003.

[12] Kimberly C. Claffy, Hans-Werner Braun, and George C. Polyzos. A parameterizable methodology for internet traffic flow profiling. *IEEE Journal of Selected Areas in Communications*, 13(8):1481–1494, 1995.

[13] Kimberly C. Claffy, George C. Polyzos, and Hans-Werner Braun. Application of sampling methodologies to network traffic characterization. *Computer Communication Review*, 23(4):194–203, 1993.

[14] John Cleary, Stephen Donnelly, Ian Graham, Anthony McGregor, and Murray Pearson. Design principles for accurate passive measurement. In *Proceedings of Passive and Active Measurement Workshop*, 2000.

[15] Irene Cozzani and Stefano Giordano. A passive test and measurement system: Traffic sampling for QoS evaluation. In *Proceedings of IEEE Global Telecommunications Conference 1998 (Globecom'98)*, volume 2, pages 1236–1241, 1998.

[16] Xing Deng. Short term behaviour of ping measurements. Master's thesis, University of Waikato, July 1999.

[17] Jack Drobisz and Kenneth J. Christensen. Adaptive sampling methods to determine network traffic statistics including the Hurst parameter. In *Proceedings of 23rd Annual IEEE Conference on Local Computer Networks (LCN'98)*, pages 238–247, 1998.

[18] Nick Duffield, Carsten Lund, and Mikkel Thorup. Charging from sampled network usage. In *Proceedings of Internet Measurement Workshop 2001 (IMW'01)*, pages 245–256, San Francisco, November 2001.

[19] Nick Duffield, Carsten Lund, and Mikkel Thorup. Properties and prediction of flow statistics from sampled packet streams. In *Proceedings of Internet Measurement Workshop 2002 (IMW'02)*, pages 159–171, Marseille, France, November 2002.

[20] Nick Duffield, Carsten Lund, and Mikkel Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of ACM SIGCOMM 2003*, pages 325–336, Karlsruhe, Germany, August 2003.

[21] Ashok Erramilli and Jonathan L. Wang. Monitoring packet traffic levels. In *Proceedings of IEEE Global Telecommunications Conference 1994 (Globecom'94)*, pages 274–280, San Francisco, November 1994.

[22] Cristian Estan, Stefan Savage, and George Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proceedings of ACM SIGCOMM 2003*, pages 137–148, Karlsruhe, Germany, August 2003.

[23] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems*, 21(3):270–313, August 2003.

[24] Wu-Chun Feng, Jeffrey R. Hay, and Mark K. Gardner. MAGNeT: Monitor for Application-Generated Network Traffic. In *Proceedings of the 10th International Conference on Computer Communications and Networks (IC3N01)*, pages 110–115, October 2001.

[25] Chuck Fraleigh, Christophe Diot, Bryan Lyles, Sue Moon, Phillipe Owezarski, Dina Papagiannaki, and Fouad Tobagi. Design and development of a passive monitoring infrastructure. In *Proceedings of Proceedings of the Thyrrhenian International Workshop on Digital Communications*, pages 556–575, 2001.

[26] Jean-Loup Gailly and Mark Adler. zlib compression library. http://www.gzip.org/zlib.

[27] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, second edition, 2003.

[28] John Shapley Gray. *Interprocess Communications in Linux: The Nooks and Crannies*. Prentice Hall PTR, 2003.

[29] Cheng Guang, Gong Jian, and Ding Wei. A traffic sampling model for measurement using packet identification. In *Proceedings of IEEE International Conference on Networks 2002 (ICON2002)*, pages 409–413, Singapore, 2002.

[30] Cameron Hughes and Tracey Hughes. *Parallel and Distributed Programming Using C++*. Addison Wesley, 2003.

[31] Gianluca Iannaccone, Christophe Diot, Ian Graham, and Nick McKeown. Monitoring very high speed links. In *Proceedings of Internet Measurement Workshop 2001 (IMW'01)*, pages 267–271, San Francisco, November 2001.

[32] Van Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, February 1990.

[33] Michael Lucas. *Absolute BSD: The Ultimate Guide to FreeBSD*. No Starch Press, 2002.

[34] Matthew J. Luckie, Anthony J. McGregor, and Hans-Werner Braun. Towards improving packet probing techniques. In *Proceedings of Internet Measurement Workshop 2001 (IMW'01)*, pages 145–150, San Francisco, November 2001.

[35] Yun Mao, Kang Chen, Dongsheng Wang, and Weimin Zheng. Cluster-based online monitoring system of web traffic. In *Proceedings of the Third International Workshop on Web Information and Data Management*, pages 47–53, 2001.

[36] Afzal Mawji and Ajit Singh. Load balanced exhaustive network data processing with a parallel cluster. In *Proceedings of the 2004 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'04)*, Las Vegas, June 2004.

[37] Mark Milward, José Luis Núnez, and David Mulvaney. Design and implementation of a lossless parallel high–speed data compression system. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):481–490, June 2004.

[38] Markus F.X.J. Oberhumer. LZO data compression library. http://www.oberhumer.com/opensource/lzo, 2002.

[39] Konstantina Papagiannaki, Rene Cruz, and Christophe Diot. Network performance monitoring at small time scales. In *Proceedings of Internet Measurement Workshop 2003 (IMW'03)*, pages 295–300, 2003.

[40] Vern Paxson, Jamshid Mahdavi, Andrew Adams, and Matt Mathis. An architecture for large-scale internet measurement. *IEEE Communications*, 36(8):48–54, August 1998.

[41] Kay A. Robbins and Steven Robbins. *Unix Systems Programming: Communication, Concurrency, and Threads*. Prentice Hall PTR, 2003.

[42] Claude Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, October 1949.

[43] Robin Sommer and Anja Feldmann. NetFlow: Information loss or win? In *Proceedings of Internet Measurement Workshop 2002 (IMW'02)*, pages 173–174, Marseille, France, November 2002.

[44] W. Richard Stevens. *UNIX Network Programming, Networking APIs: Sockets and XTI*, volume 1. Prentice Hall PTR, second edition, 1998.

[45] International Telecommunication Union. Recommendation v.42 bis. http://www.itu.int.

[46] International Telecommunication Union. Recommendation v.44. http://www.itu.int.