Report Number: WUCSE-2004-72

2004-12-01

Techniques for Processing TCP/IP Flow Content in Network Switches at Gigabit Line Rates

David Vincent Schuehler

The growth of the Internet has enabled it to become a critical component used by businesses, governments and individuals. While most of the traffic on the Internet is legitimate, a proportion of the traffic includes worms, computer viruses, network intrusions, computer espionage, security breaches and illegal behavior. This rogue traffic causes computer and network outages, reduces network throughput, and costs governments and companies billions of dollars each year. This dissertation investigates the problems associated with TCP stream processing in high-speed networks. It describes an architecture that simplifies the processing of TCP data streams in these environments and presents a... Read complete abstract on page 2.

Techniques for Processing TCP/IP Flow Content in Network Switches at Gigabit Line Rates

David Vincent Schuehler

Complete Abstract:

The growth of the Internet has enabled it to become a critical component used by businesses, governments and individuals. While most of the traffic on the Internet is legitimate, a proportion of the traffic includes worms, computer viruses, network intrusions, computer espionage, security breaches and illegal behavior. This rogue traffic causes computer and network outages, reduces network throughput, and costs governments and companies billions of dollars each year. This dissertation investigates the problems associated with TCP stream processing in high-speed networks. It describes an architecture that simplifies the processing of TCP data streams in these environments and presents a hardware circuit capable of TCP stream processing on multi-gigabit networks for millions of simultaneous network connections. Live Internet traffic is analyzed using this new TCP processing circuit.

2004-72

Techniques for Processing TCP/IP Flow Content in Network Switches at Gigabit Line Rates, Doctoral Dissertation, December 2004

Authors: Schuehler, David V.

Abstract: The growth of the Internet has enabled it to become a critical component used by businesses, governments and individuals. While most of the traffic on the Internet is legitimate, a proportion of the traffic includes worms, computer viruses, network intrusions, computer espionage, security breaches and illegal behavior. This rogue traffic causes computer and network outages, reduces network throughput, and costs governments and companies billions of dollars each year.

This dissertation investigates the problems associated with TCP stream processing in high-speed networks. It describes an architecture that simplifies the processing of TCP data streams in these environments and presents a hardware circuit capable of TCP stream processing on multi-gigabit networks for millions of simultaneous network connections. Live Internet traffic is analyzed using this new TCP processing circuit.

Type of Report: Other

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

TECHNIQUES FOR PROCESSING TCP/IP FLOW CONTENT IN NETWORK SWITCHES AT GIGABIT LINE RATES

by

David Vincent Schuehler

Prepared under the direction of Professor John W. Lockwood

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

December, 2004

Saint Louis, Missouri

Washington University in St. Louis
Technical Report WUCSE-2004-72
Nov 22, 2004
http://www.arl.wustl.edu/arl/projects/fpx/reconfig.htm

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

TECHNIQUES FOR PROCESSING TCP/IP FLOW CONTENT
IN NETWORK SWITCHES AT GIGABIT LINE RATES

by David Vincent Schuehler

ADVISOR: Professor John W. Lockwood

December, 2004
Saint Louis, Missouri

The growth of the Internet has enabled it to become a critical component used by businesses, governments and individuals. While most of the traffic on the Internet is legitimate, a proportion of the traffic includes worms, computer viruses, network intrusions, computer espionage, security breaches and illegal behavior. This rogue traffic causes computer and network outages, reduces network throughput, and costs governments and companies billions of dollars each year.

This dissertation investigates the problems associated with TCP stream processing in high-speed networks. It describes an architecture that simplifies the processing of TCP data streams in these environments and presents a hardware circuit capable of TCP stream processing on multi-gigabit networks for millions of simultaneous network connections. Live Internet traffic is analyzed using this new TCP processing circuit.

This dissertation is dedicated to

my family,

my friends,

and the pursuit of knowledge.

Contents

List of Tables

List of Figures

Abbreviations

AAL	ATM Adaptation Layer
ALU	Arithmetic Logic Unit
ARL	Applied Research Laboratory
ARQ	Automatic Repeat Request
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
BGP	Border Gateway Protocol
CAM	Content Addressable Memory
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CPU	Central Processing Unit
DDR	Double Data Rate (Memory)
DMA	Direct Memory Access
DoS	Denial of Service
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FIFO	First-In First-Out queue
FIPL	Fast Internet Protocol Lookup

FPGA	Field Programmable Gate Array
FPX	Filed-programmable Port Extender
FSM	Finite State Machine
HEC	Header Error Checksum
HTTP	Hyper Text Transfer Protocol
IP	Internet Protocol
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
iSCSI	Internet SCSI
ISP	Internet Service Provider
LC	Logic Cell
MAC	Media Access Control
MIB	Management Information Base
MTU	Maximum Transmission Unit
MRTG	Multi Router Traffic Grapher
NAS	Network Attached Storage
NIC	Network Interface Card
OC	Optical Carrier
PHY	Physical Layer
RAM	Random Access Memory
RFC	Recursive Flow Classification
RISC	Reduced Instruction Set Computer
SAN	Storage Area Network

SAR	Segmentation and Reassembly
SCSI	Small Computer Systems Interface
SDRAM	Synchronous Dynamic Random Access Memory
SNMP	Simple Network Management Protocol
SRAM	Static Random Access Memory
SSL	Secure Sockets Layer
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TOE	TCP Offload Engine
UTOPIA	Universal Test and Operations PHY Interface for ATM
VCI	Virtual Channel Identifier
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit
VPI	Virtual Path Identifier
WUGS	Washington University Gigabit Switch
ZBT	Zero Bus Turnaround (memory devices)

Acknowledgments

First and foremost, I would like to thank my parents, Jerry and Lois Schuehler. They have instilled in me a strong work ethic, a thirst for knowledge, and the confidence to achieve any goal. Without their love, support, and guidance, none of my accomplishments would have been possible.

I would like to thank my brother Chris for his friendship and companionship over the years. He has taught me many things and provided me the opportunity to participate in a variety of activities. We have had yearly excursions out west, rock climbing, back packing, mountain biking, snow skiing, hiking and camping.

I would also like to thank my sister Nancy and brother-in-law Jeff for helping me during my doctoral studies. Due to their proximity, they were called upon time and time again to take care of my dog, Fritz, while I was working late in the lab. They also provided me with a nephew, Nathan, who managed to brighten my days and offered a much needed distraction from school work.

Additional thanks goes out to Bertrand Brinley who wrote the *The Mad Scientists' Club* books [15, 16]. These books captivated me as a child and introduced me to the wonderful adventures awaiting those with an understanding of math, science and engineering.

I would also like to thank Scott Parsons, Don Bertier, Andy Cox, and Chris Gray for writing letters of recommendation. These recommendation letters were very flattering of my intellect and abilities, and continue to be a great source of pride for me. I do not believe that I would have been accepted into the doctoral program at Washington University without their compelling letters of recommendation.

Thanks also goes out to the Computer Science Department graduate committee at Washington University in St. Louis for accepting me into the doctoral program. They provided me with the opportunity to prove myself capable and accomplish this goal.

I would also like to thank all those at Reuters (formerly Bridge Information Systems) who supported my efforts to pursue a doctoral degree. My supervisor Scott Parsons, senior site officer Deborah Grossman, and coworker John Leighton deserve additional thanks for going out of their way to make this doctoral work possible.

James Hartley has been a friend and colleague since my freshman year of college in 1983. James and I share a passion for technical topics in the computer and networking fields. We spent many hours on the phone discussing various aspects of my graduate studies. His input as an impartial and independent party has been invaluable.

A special thanks goes out to Tanya Yatzeck who tirelessly reviewed much of my written work. In addition, Tanya nominated me for the St. Louis Business Journal 2004 Technology Award. The ensuing award and accompanying accolades would not have been possible if Tanya hadn't taken the initiative.

I would like to thank Jan Weller and Steve Wiese of Washington University Network Technology Services for providing live Internet traffic feeds which were used in the debugging and analysis phases of this research project. This network traffic was analyzed using the hardware circuits described in this dissertation.

No doctoral research project occurs in a vacuum, and this research is no exception. My interactions with the faculty, staff and students of the department of Computer Science and Engineering and the Applied Research Laboratory were invaluable in helping me complete this research work. I would especially like to thank Dr. Jon Turner, Dr. Ron Citron, Fred Kuhns, John DeHart, Dave Taylor, James Moscola, Dave Lim, Chris Zuver, Chris Neeley, Todd Sproull, Sarang Dharmapurikar, Mike Attig, Jeff Mitchell, and Haoyu Song.

This research work was supported in part by Global Velocity. I would like to thank Matthew Kulig and Global Velocity for supporting my research work.

I would like to thank the members of my thesis committee: Dr. John Lockwood, Dr. Chris Gill, Dr. Ron Loui, Dr. David Schimmel, and Dr. Ron Indeck.

I would like to give special thanks to my research advisor, Dr. John Lockwood, for his invaluable guidance and direction during my doctoral studies. He was a sounding board for thoughts and ideas and helped spur research in new directions.

Finally, I would like to thank my dog Fritz, who had to deal with a greatly reduced play schedule. He also had to endure long days and nights alone while I was working on the computer, doing research, writing papers, reading papers and doing homework.

Thank you all.

David Vincent Schuehler

Washington University in Saint Louis
December 2004

Preface

User application data moves through the Internet encapsulated in network data packets. These packets have a well-defined format containing several layered protocol headers and possibly trailers which encapsulate user data. The vast majority of Internet traffic uses the Transmission Control Protocol (TCP). The TCP implementation on the sending host divides user application data into smaller transmission segments and manages the delivery of these segments to remote systems. The protocol stack on the remote system is responsible for reassembling the segments back into the original data set, where it is presented to the user or the application.

Extensible networking services, like those that detect and eliminate Internet worms and computer viruses as they spread between machines, require access to this user application data. These services need to process the various protocols in order to reassemble application data, prior to performing any detection or removal operations. TCP processing operations similar to those which occur in protocol stacks of end systems are necessary.

Large, high-speed routers and switches are used to route data through the interior of the Internet because they are capable of handling traffic from millions of end systems at multi-gigabit per second data rates. To operate within this environment, extensible networking services need to process millions of packets per second. Existing services which block Internet worms and computer viruses do not have sufficient throughput to operate on a network backbone.

This dissertation investigates the problems associated with high performance TCP processing systems and describes an architecture that supports flow monitoring in extensible networking environments at multi-gigabit per second data rates. This architecture provides stateful flow tracking, TCP stream reassembly services, context storage, and flow manipulation services for applications which monitor and process TCP data streams. An FPGA-based implementation of this architecture was used to analyze live Internet traffic.

Chapter 1

Introduction

Studies show that over 85% of the network packets on the Internet utilize Transmission Control Protocol (TCP) [112]. Content-aware networking services, like data security and content scanning, need to look inside of TCP flows in order to implement higher levels of functionality. In order to accomplish this, the TCP processing must be capable of reconstructing data streams from individual network packets. TCP processing systems capable of achieving this in gigabit speed networking environments do not currently exist.

New protocol processing systems are necessary to satisfy the current and future needs of content-aware network services. These services will have a greater impact on network traffic when located with large routers processing traffic on high-speed network connections exceeding 2Gbps. These types of routers are also most likely to contain extensible networking technology.

The concept of extensible networking implies that reconfigurable and reprogrammable elements exist within the network which can be altered, or extended, during the normal operation of the network. Extensible networking switches may employ application-specific integrated circuits (ASICs), field-programmable gate arrays (FPGAs), network processors, or general purpose microprocessors which can be programmed or configured to operate on the network traffic which transits the switch. Extensible networking environments can enhance the end user's experience by incorporating value-added services into the network infrastructure. In order to be effective, services which operate in these environments require access to the application level data which traverses the network. This research involves the design and implementation of a TCP processing system which makes application data available to high-speed network services. A simple, hardware-based application interface provides the means by which data processing circuits can easily access the payload contents of TCP flows.

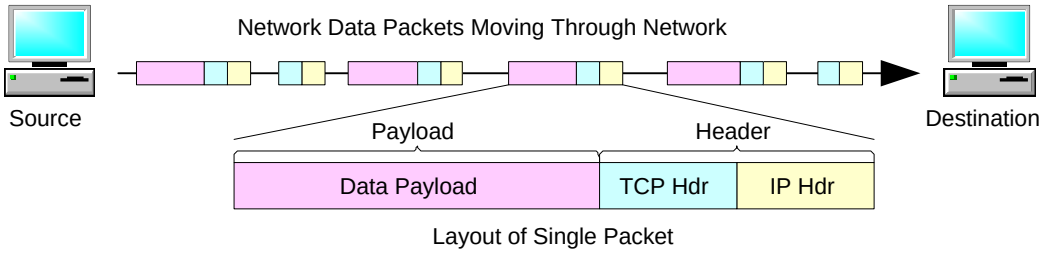


Figure 1.1: Anatomy of a Network Packet

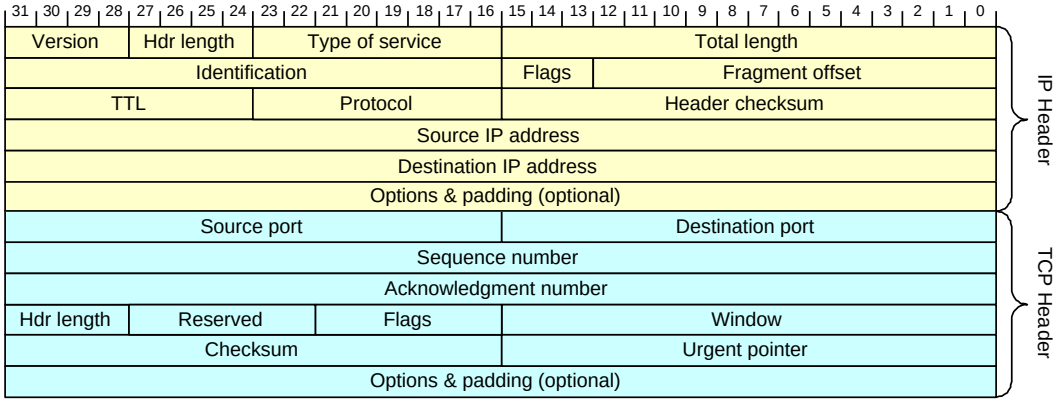


Figure 1.2: IP(v4) and TCP Headers

Data in TCP flows moves through the Internet encapsulated in network packets. Every packet contains a header section and possibly a payload section. The header section can be further subdivided to provide information about encapsulated protocols. Figure 1.1 shows the layout of TCP packets moving through the network. Each packet contains an Internet Protocol (IP) header, a TCP header, and possibly a data payload section.

To exchange information over a network, a TCP connection is established between two end systems. Information placed in the payload section of one or more packets is delivered to the remote system. As these packets traverse the Internet, they can be dropped, reordered, duplicated, or fragmented while they make their way to the destination host. It is the responsibility of the TCP implementation on the end systems to provide end-to-end reordering and retransmission services.

Figure 1.2 shows a diagram of the header fields for TCP and IP version 4. The IP header fields include packet routing information which is used by network equipment to deliver packets to the proper end destination. The information contained in the IP header is similar in function to the address on traditional courier-based mail. The TCP header fields

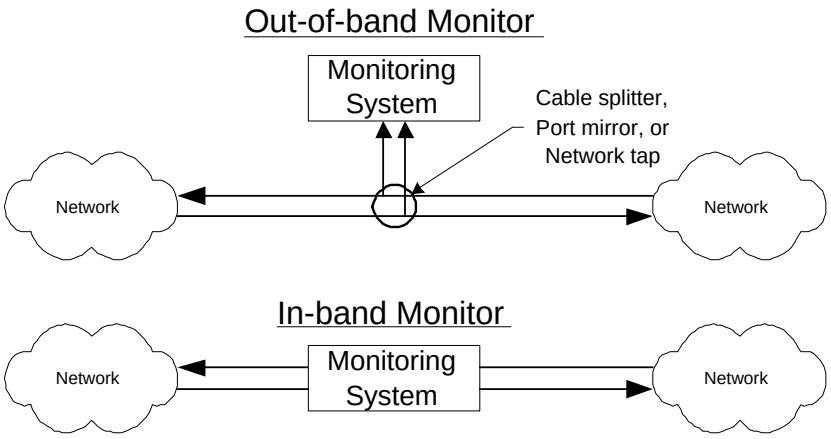


Figure 1.3: Types of Network Monitors

provide information needed to reconstruct data streams from individual network packets. In addition, these header fields contain data needed to perform connection setup and teardown, packet retransmissions, and localized packet delivery information.

Network monitors can be classified as either out-of-band or in-band, as illustrated in Figure 1.3. An out-of-band network monitor is a passive device and is typically implemented with a cable splitter, a network tap, port replication or other content duplicating technique which delivers a copy of the network traffic to the monitor. Passive monitors have no means by which to alter network content. In contrast, an in-band network monitor resides within the data path of the network and can alter network traffic by either inserting, dropping, or modifying packets. As an in-band network monitor, the TCP processing system presented in this dissertation provides a flexible platform upon which either passive or active extensible networking services can be developed.

Processing higher level protocols, such as TCP, is a complex task. This complexity is increased when operating on multi-gigabit links in high-speed networking environments, potentially dealing with millions of packets per second and processing millions of active TCP connections in the network at any one time. Data monitoring and processing applications are varied and have a broad range of resource requirements. Simple applications like those that count data bytes or represent traffic patterns may have relatively small implementations. In contrast, applications which detect and eliminate many different types of Internet worms and viruses may utilize substantially more resources. Specialized applications which implement custom network security features may also require complex processing operations for every packet on the network. Limitations in chip densities prevent

these large, complex applications from being completely implemented in a single device. A system using multiple processing devices would support these complex applications. In order for this type of design to work, efficient mechanisms are required for coordinating tasks amongst the multiple devices.

This dissertation presents a hardware circuit called the TCP-Processor which is capable of reassembling TCP data flows into their respective byte streams at multi-gigabit line rates. The reference implementation contains a large per-flow state store which supports 8 million bidirectional TCP flows. Additional logic enables the payload contents of a TCP flow to be modified. To support complex flow processing applications, a flexible and extensible data encoding system has been developed which enables data passing and task coordination between multiple devices. This technology enables a new generation of content-aware network services to operate within the core of the Internet.

1.1 Problem Framework

A large heterogenous network, such as the Internet, provides network connectivity between a vast number of end systems and autonomous environments. Figure 1.4 shows a representative network topology. The interior of the network is made up of core routers - responsible for routing data through the interior of the network - and gateway routers. Each gateway router provides a single point of entry into the network for an external organization. Many different types of organizations, including government agencies, corporations, municipalities, and universities connect to the network by interfacing with a gateway router. Cell phones and handheld computers connect to the network through cellular towers to carriers which provide access to the Internet. In addition, Internet Service Providers (ISPs) provide communication services to individual and organizational customers, and satellite and fiber-optic links interconnect wide-area networks.

Due to the vast number of end systems which communicate over the Internet and the relatively small number of routers which forward traffic in the interior of the network, communication between these routers occurs at a very high bandwidth. Currently, interconnects typically operate over communication links ranging in speed from OC-3 to OC-768. Network routers capable of communicating at 160 Gigabits/second are expected in the future. Table 1.1 compares the various types of communication links, their corresponding data rates, and the rate at which packets of different sizes can be transmitted over those links.

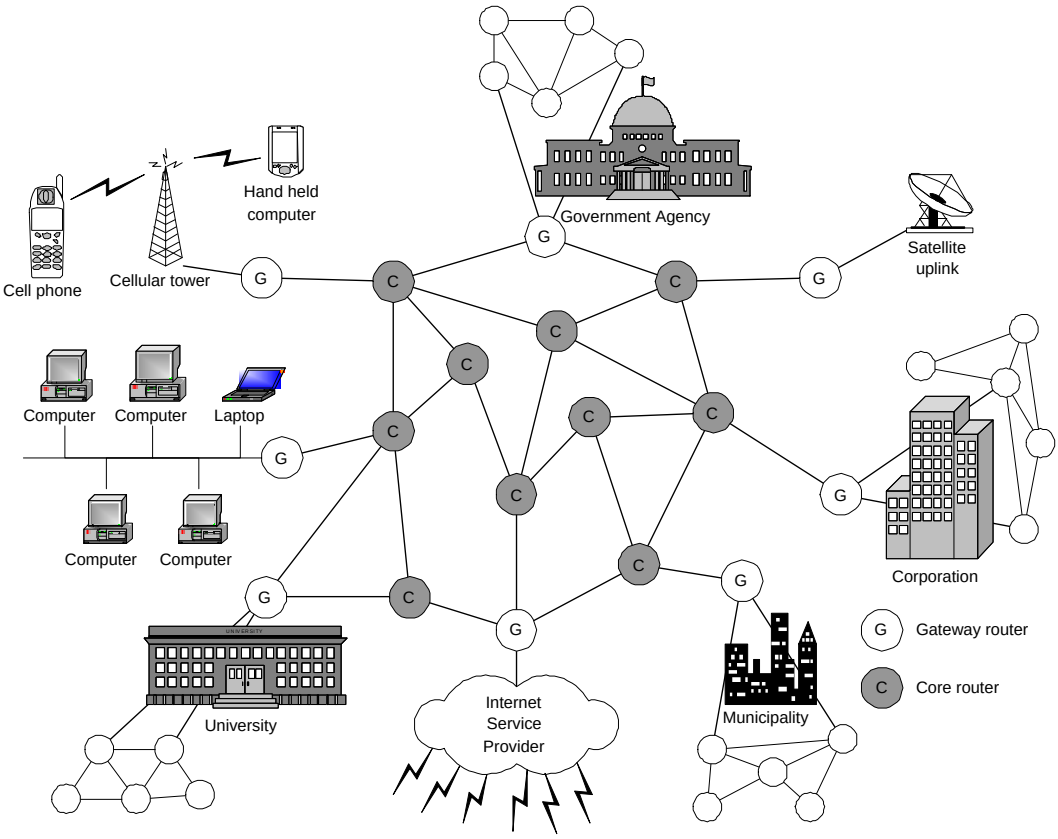


Figure 1.4: Heterogenous Network

1.2 Problem Statement

There are distinct advantages associated with consolidating common network services. In addition to running data security, data compression, intrusion prevention, and virus elimination applications on end systems, applications can be moved into the network at gateway routers. Running these applications in the network reduces the administrative overhead of managing the number of installation points. Large organizations may have thousands of end systems to manage. Instantly updating virus filters on all end systems in the face of a new worm or virus outbreak can be a daunting task. In contrast, updating a few well-placed network appliances positioned within the network can provide protection from the virus outbreak in a fraction of the time required to update all end systems.

Universities and Internet Service Providers (ISPs) have networks where the network administrators do not have access to or control of the machines connected to the network. Student computers and client ISP systems are connected to these networks, but the network

Table 1.1: Optical links and associated data rates

Link type	Data rate	40 byte pkts/sec	64 byte pkts/sec	500 byte pkts/sec	1500 byte pkts/sec
OC-3	155 Mbps	.48 M	.3 M	38 K	12 K
OC-12	622 Mbps	1.9 M	1.2 M	.16 M	52 K
GigE	1.0 Gbps	3.1 M	2.0 M	.25 M	83 K
OC-48	2.5 Gbps	7.8 M	4.8 M	.63 M	.21 M
OC-192/10 GigE	10 Gbps	31 M	20 M	2.5 M	.83 M
OC-768	40 Gbps	125 M	78 M	10 M	3.3 M
OC-3072	160 Gbps	500 M	312 M	40 M	13 M

operations staff has no ability to patch or upgrade the end systems. In these types of environments, data security, intrusion detection, and worm and virus preventions systems have to occur within the fabric of the network. Furthermore, with the widespread acceptance of cellular and wireless technologies, computer systems can quickly appear and disappear from the network, only to reappear later in a different part of the network.

Data security applications which prevent the dissemination of proprietary or confidential documents to outside parties are ideally suited for placement at the interface to the external network. This type of security solution would allow confidential and proprietary documents to be passed freely amongst computers within an organization, but prevents their transmission to non-trusted systems outside the organization.

Most existing network monitoring systems which perform data security and intrusion prevention tasks are software-based. These systems have performance limitations which prevent them from operating within high-speed networked environments. The research associated with this dissertation addresses the performance limitations associated with existing network services.

This dissertation presents an architecture for a high-speed TCP processing system which provides access to TCP stream content at locations within the network. The architecture is capable of processing traffic at multi-gigabit per second data rates and supports the monitoring and processing of millions of active TCP flows in the network.

1.3 Contributions

The main focus of this research involves the design, development, and implementation of hardware circuits which are able to process TCP flows within the context of multi-gigabit per second communication links. The primary contributions of this research are:

- The definition of a high-performance TCP flow processing architecture, called TCP-Processor, capable of processing large numbers of active TCP connections traversing multi-gigabit network links. A hardware implementation of this architecture supports the processing of 8 million simultaneous TCP connections on a network link operating at 2.5Gbps.

- An architecture that coordinates high-performance packet processing circuits operating on separate devices by defining an inter-device transport protocol. This protocol is extensible, which allows for future enhancements and extensions without requiring modifications to the underlying protocol specification. The self-describing nature of the protocol allows previous versions of the processing circuits to work with newer extensions which do not pertain to that specific component.

- An analysis of live Internet traffic processed by hardware circuit implementations of the TCP processing architecture. Network traffic travelling between the Washington University campus network and the Internet was processed by the TCP flow monitoring circuits outlined in this dissertation. Statistics collected while processing this traffic are analyzed and presented.

Additional contributions presented in this dissertation include the development of two sample TCP flow monitoring applications, PortTracker and Scan, which illustrate how to interface with the TCP-Processor to perform TCP flow processing on large numbers of active flows at multi-gigabit data rates. Furthermore, it defines a common statistics packet format which is used by the TCP flow processing circuits and by other researchers at the Applied Research Laboratory. It also presents extensible data collection and charting utilities which have been developed to aid in the collection and presentation of the monitored statistics. Detailed documentation on the design, layout, and operation of the various TCP processing circuits is provided. The appendices include setup, configuration, and usage instructions so that other investigators have the information necessary to either reproduce or extend this research.

1.4 Organization of Dissertation

Chapter 1 provides introductory material and an overview of the problem domain. The problem framework and a problem statement are discussed along with the contributions of the dissertation and the organization of the dissertation.

Chapter 2 provides background and motivation information for the dissertation. It discusses the currently available computing platforms along with their relative merits for use in high-performance network data processing systems. This chapter also covers a thorough review of the challenges associated with protocol processing in high-speed networks. Chapter 3 examines related work in network monitoring and processing systems. It presents a taxonomy of network monitors to show the problem domain of the research relating to this dissertation. It also discusses other software-based and hardware-based network monitors, along with their features and shortcomings. In addition, this chapter presents current research in packet classification techniques and covers information on other types of network devices. These devices include load balancers, Secure Sockets Layer (SSL) accelerators, intrusion detection systems and TCP offload engines; all of which require protocol processing operations similar to those presented in this dissertation. Finally, this chapter presents current research involving hardware-accelerated content scanners to showcase a class of applications which are ideal candidates for integration with the TCP-Processor.

Chapter 4 provides an overview of the TCP processing architectures developed as part of this research. Initial investigations into high-performance TCP processing in hardware circuits led to the development of the TCP-Splitter technology. The knowledge gained through this research contributed to the development of the TCP-Processor technology. This chapter describes both of these architectures.

Chapter 5 provides information regarding the testing environment where the TCP-Processor was developed, refined, and analyzed. This environment includes hardware platforms, circuit designs, and control software. This chapter discusses each of the components of this environment.

Chapter 6 provides detailed descriptions of the implementation and internal operation of the TCP-Processor circuit. Implementation details of each circuit component are discussed separately, providing insight into the operation and interaction among the various components. This chapter also describes the lightweight client interface which provides easy access to TCP stream content for large numbers of TCP connections in an efficient manner.

Chapter 7 examines the StreamExtract circuit. This circuit forms the basis for all multi-device TCP processing designs. This chapter includes a description of the data encoding and decoding techniques which allow annotated network traffic to pass easily between devices. It describes the encoded data format, highlighting the extensible and expandable features of the protocol.

Chapter 8 discusses the TCP-Lite Wrappers which provide access to TCP stream content in multi-device TCP processing environments. These wrappers include lightweight hardware circuits which decode and re-encode TCP stream traffic. This includes exposure of a client interface which is identical to the interface of the TCP-Processor. In addition to the TCP-Lite Wrappers, this chapter presents PortTracker and Scan as example circuits which use this wrapper to perform TCP flow processing. The PortTracker circuit keeps track of the number of packets sent to or received from various well known TCP ports. The Scan circuit searches TCP stream content for up to four 32-byte signatures.

Chapter 9 analyzes live Internet traffic using the TCP flow processing technology presented in this dissertation. It consists of the examination of the Internet traffic for the Washington University campus network during a five week period.

Chapter 10 summarizes the work and contributions of this dissertation and provides concluding remarks. It presents information regarding the lessons learned while processing live Internet traffic along with information on some of the tools developed to aid in the debugging process. Finally, it presents information on the utility of the TCP-Processor.

Chapter 11 describes future directions for high-performance TCP processing research. This includes extensions to the TCP-Processor, increasing the current capabilities and performance. It also covers integration with TCP flow processing applications.

Appendix A describes usage information for the various TCP processing circuits. It provides specific information on how to build, configure and use the TCP processing circuits described in this dissertation. This information will enable other researchers to easily reproduce and validate the results of this research and provide a starting point for performing other advanced TCP flow processing research in high-speed networking environments.

Appendix B discusses several different methods of capturing traffic and generating simulation input files. It describes several hardware circuits and software-based tools developed as part of this research to provide the functional debugging and verification tools required to validate the operation and performance of the TCP Processor. These include hooks added to all of the TCP processing circuits which allow network traffic and associated context data to be stored into external memory devices. In addition, this appendix

includes a description of software routines that can extract and format this data from memories so it can be used easily by the hardware circuit simulation tool, ModelSim. Finally, this appendix describes the operation and use of these hardware and software data capturing and formatting tools.

Appendix C describes the statistics gathering and charting routines used in conjunction with the circuits described in this dissertation. A StatsCollector application collects statistics information generated by the various TCP processing circuits and writes the data to daily disk files. A Simple Network Management Protocol (SNMP) sub-agent captures this same statistics information, accumulates ongoing event counters, and republishes the data as SNMP variables which can be accessed easily using industry standard network management tools. A configuration file for the Multi Router Traffic Grapher (MRTG) commands MRTG to generate daily, weekly, monthly and yearly charts of each of the statistics values produced by the TCP processing circuits.

Appendix D contains additional traffic charts obtained by analyzing the Washington University Internet traffic. The charts examine traffic collected on August 27th, 2004 and September 16th, 2004.

Chapter 2

Background and Motivation

Existing network monitors are unable to both (1) operate at the high bandwidth rates and (2) manage the millions of active flows found in today's high-speed networks. Instead, network monitors typically perform monitoring activities on end systems or in local area networks where the overall bandwidth and the total number of flows to be processed is low. In the future, many other types of network services will require access to the TCP stream data traversing high-speed networks. Such services may include the following applications:

- Worm/virus detection and removal
- Content scanning and filtering
- Spam detection and removal
- Content-based routing
- Data security
- Data mining
- Copyright protection
- Network monitoring

The research outlined in this dissertation enables these services to operate at multi-gigabit speeds by defining and implementing a hardware-based architecture capable of high-performance TCP processing. The resulting architecture provides a flexible environment in which any high performance data processing application can be implemented.

A service which detects and prevents the spread of Internet worms and computer viruses is ideally suited for this technology. Since worms and viruses spread very quickly, prevention techniques which involve human interaction are essentially ineffective. Humans are not capable of reacting quickly enough to stop worm and virus attacks from spreading.

These types of attacks require automated prevention techniques for containment and prevention of widespread infections [140]. In addition, the direct costs associated with these attacks could negatively affect the global economy. Table 2.1 provides analysis of the costs associated with several recent attacks on the Internet.

Table 2.1: Cost of Internet Attacks [22, 86, 5, 34, 56, 33, 1, 85]

Year	Worldwide Economic Impact ($ U.S.)	Representative Attacks (cost)
2003	$236 Billion	Sobig.F ($2 Billion) Blaster ($1.3 Billion) Slammer ($1.2 Billion)
2002	$118 Billion	KLEZ ($9 Billion) Bugbear ($950 Million)
2001	$36 Billion	Nimbda ($635 Million) Code Red ($2.62 Billion) SirCam ($1.15 Billion)
2000	$26 Billion	Love Bug ($8.75 Billion)
1999	$20 Billion	Melissa ($1.10 Billion) Explorer ($1.02 Billion)

Detection of intrusions embedded within TCP data flows requires stream reassembly operations prior to scanning for virus signatures. Network-based Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) typically operate by searching network traffic for digital signatures which correspond to various types of known intrusions, including worms and viruses [67]. To detect intrusions, IDS appliances must perform Deep Packet Inspections (DPI) which scan packet payloads and process packet headers [51].

In recent years, Internet-based worm and computer virus attacks have caused widespread problems to Internet users. This is due to the fact that large numbers of homogeneous computers are interconnected by high-speed network links. When attacks occur, they can rapidly reach epidemic proportions. The SoBig.F virus infected over 200 million computers within a week and accounted for more than 70% of all email traffic on Aug 20, 2003 [76]. The Code Red virus infected 2,000 new machines every minute during its peak [87]. The MSBlast worm infected more than 350,000 computers [75]. Internet worms and viruses typically spread by probing random IP addresses, searching for vulnerable computers. At its peak, the Slammer worm performed over 55 million scans per second [86].

Internet worms and computer viruses consume huge amounts of computing and networking resources and place an enormous burden on the Internet as a whole.

Moore *et al*. analyzed the behavior of self-propagating worms and viruses [88]. They performed simulations using two different containment methodologies in order to determine the relative effectiveness of each approach. The first containment methodology performed address blacklisting (or address filtering) and the second performed content scanning and filtering. They determined that the content scanning and traffic filtering technique is far superior at preventing the spread of malicious code than address blacklisting. Their results also show that content filtering is much more effective at stopping the spread of viruses when filters are distributed throughout the Internet on high-speed links between ISPs. If placed closer to the periphery of the Internet, these filters were ineffective in stopping widespread infection from worms and viruses. Their results highlight the need for devices capable of high performance content filtering and virus containment. The authors of [88] also lament the fact that virus prevention systems of this nature are not currently available.

Several problems currently limit the effectiveness of monitoring end systems for viruses. Popular operating systems, such as Windows, consist of a large volume of software that is often in need of security updates and patches. The end user must diligently apply updates in a timely manner since network-based attacks can be initiated within days of a vulnerability being detected. For a large institution which may have thousands or tens of thousands of end systems at many locations, the task is even more complex. Installing patches and software updates is typically a serial operation, so the time required to distribute new software or configuration information scales linearly with the number of computers that need to be updated. In addition, a central authority may not have complete access to control all of the machines operating in the network. Even if it did, a security hole remains when, for example, a computer is powered off during an update cycle and fails to receive the new version of the software. Universities and Internet Service Providers (ISPs) represent two types of networks where the network administrator does not have administrative control over machines connected to the network. Further, the proliferation of laptop computers, wireless networks and dynamic addressing allow computers to have great mobility which makes network management even harder. Laptops can appear and disappear from the network, only to reappear a short time later at different locations in the network. This complicates the task of securing networks from attacks.

Monitoring flows in high performance networks requires new methodologies of stream scanning. Moore's Law predicts that the number of transistors that fit on a chip

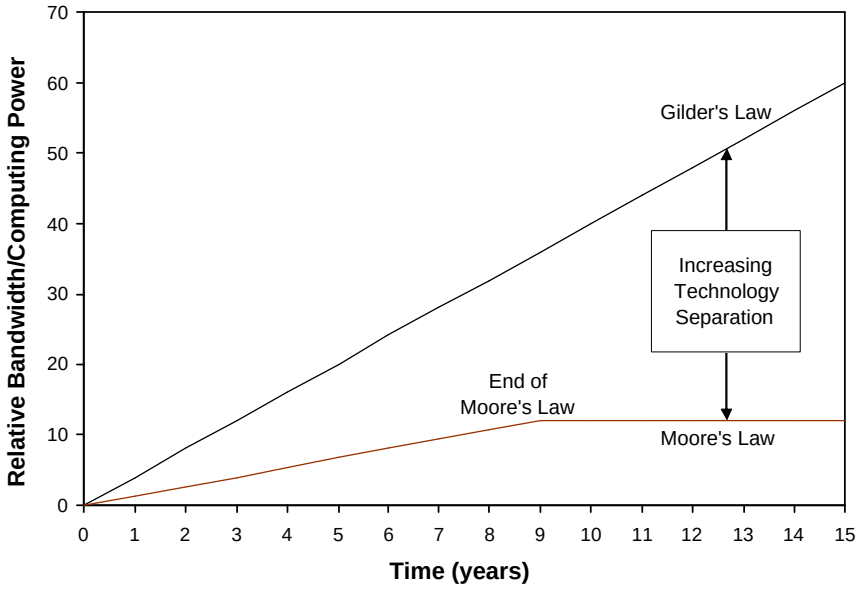


Figure 2.1: Gilder's Law versus Moore's Law

will double every eighteen months [89]. Since transistor count is directly related to computing power, this law implies that computing power will double every eighteen months. This exponential gain is not sufficient to keep pace with advances in communication technology. Gilder's Law of Telecom predicts that bandwidth will grow at least three times faster than computing power [44]. As time progresses, it is reasonable to expect that the gap between network bandwidth and computing power will continue to widen. Moore's Law will cease to hold within the next 10 years due to quantum effects which occur when transistor gate sizes approach 10 nanometers, further exacerbating this gap. The discrepancy between network bandwidth and processing power can clearly be seen in Figure 2.1.

2.1 Hardware Processing Technologies

Three classes of processing technologies can be used to satisfy the computational needs of a high performance TCP processing system. The first is a microprocessor, the second is an Application Specific Integrated Circuit (ASIC), and the third is a Field Programmable Gate Array (FPGA). Network processors are considered a subtype classification of microprocessors. Although these processors have been customized for networking applications, they have performance limitations similar to general purpose processors.

2.1.1 Microprocessors

The current generation of general purpose microprocessors is capable of operating at clock frequencies of 3GHz. There is an important distinction, however, between clock frequency and the amount of work performed per unit of time. The ratio of the number of instructions executed versus the number of clock cycles required to execute those instructions ($\frac{instructions}{clock\ cycle}$) provides an indication of the amount of work performed per unit of time. For older Complex Instruction Set Computers (CISC), this ratio was less than one, indicating that multiple clock cycles were required to complete the processing for a single operation. In the 1980s, Reduced Instruction Set Computers (RISC) were developed to increase this ratio to 1 such that an operation was completed every clock cycle. The instruction set of a RISC machine typically contains a load and store type architecture with fewer memory addressing modes. In general, a RISC machine requires more instructions than a CISC machine when performing the same amount of work [28].

Pipelining a processor core is a common technique for increasing the amount of work completed per unit of time. By breaking the execution of an instruction into multiple stages (such as fetch, decode, execute, and store), microprocessors operate at higher clock frequencies while still performing the same amount of work per clock cycle. Depending on the instruction sequence, a pipelined architecture can execute one instruction per clock cycle at a much higher clock frequency than can be achieved when there is no pipelining. There are disadvantages associated with a pipelined architecture. The efficiency of a pipeline is not realized until the pipeline is full of instructions. Prior to this point, no work is accomplished for several clock cycles while the pipeline is being primed. Other inefficiencies arise from context switches [97], pipeline stalls, and mispredicted branches [11]. These events cause caches and pipelines to be flushed, which in turn accounts for additional clock cycles where no work is completed. In addition, the more stages there are to a pipeline, the larger the impact of flushing and refilling the pipeline. Super-pipelined architectures [113] further increase the number of pipeline stages by taking long or complex operations and breaking them into several smaller steps. This supports even faster clock frequencies over normal pipelined architectures. Breaking the previous example into the following eight stages is an example of super-pipelining: fetch1, fetch2, fetch3, decode, execute1, execute2, store1, and store2.

Superscalar processing technology [116] allows multiple operations to be performed on each clock cycle which further increases the work per clock cycle. Superscalar architectures have multiple execution units within the core of the microprocessor. Processors which

contain multiple execution units are less likely to keep all execution units busy doing useful work, thus reducing the amount of real work performed at each clock cycle.

Regardless of the caching subsystem, the number of pipeline stages, or the number parallel execution units, virtually every processor-based computer ever built is based upon the design principals of the von Neumann architecture [103]. This architecture defines a computer as containing a Central Processing Unit (CPU), a control unit, a memory unit, and input/output units [94]. Combined, these units read and execute a stored sequence of instructions. The Arithmetic Logic Unit (ALU) forms the core of the CPU and performs all operations which manipulate data. Since a von Neumann architecture executes a stored program, larger amounts of external memory are required for processor-based systems as compared to systems based on other hardware processing technologies. Processors also operate slower than other technologies because program instructions must be loaded from memory prior to performing any operation. In addition to program data, input data, temporary variables, intermediate data, computation results and output data are all typically stored in the same external memory devices. This can lead to congestion while accessing external memory devices.

2.1.2 Application Specific Integrated Circuits

Application Specific Integrated Circuits (ASICs) are devices which have been custom-designed to perform very specific sets of operations. These devices can contain processor cores, memories, combinatorial logic, I/O components or other specialized circuitry which are dedicated to solving a particular problem. Because an ASIC is developed for a specific application, it can be customized to give optimal performance. These devices offer superior performance over other processing technologies. They have smaller die sizes, lower heat dissipation, a smaller footprint, lower power consumption, and higher performance.

One disadvantage of ASIC devices is that a large nonrecurring engineering cost must be amortized over the total number of devices produced. This causes these devices to be extremely expensive during initial development phases, or when production runs are small. If total device counts number in the hundreds of thousands or greater, ASIC devices become the most affordable hardware processing technology.

2.1.3 Field Programmable Gate Arrays

Field Programmable Gate Array (FPGA) devices offer another hardware platform for networking application development. They contain a large matrix of logic blocks, memory

blocks and Input/Output (I/O) blocks which are interconnected via a programmable signal routing matrix. An FPGA logic block has multi-input look-up tables which can be dynamically configured to implement a variety of boolean logic operations. Each Configurable Logic Block (CLB) in a Xilinx Virtex E series FPGA has four Logic Cells (LC) [137]. Each LC contains a 4-input look-up table, an additional carry chain logic, and a 1-bit storage element. A CLB can be configured to implement any boolean logic operation with 5 inputs. FPGAs typically contain additional logic to manage and transport multiple clock domains through the switch fabric. [139] provides an overview of FPGA technology and implementation details.

A Hardware Description Language (HDL) such as Verilog HDL or Very Large Scale Integrated Circuit (VLSIC) HDL (VHDL) produces either a behavioral or structural description of the intended circuit operation. This description language is compiled down into a configuration file that configures the reprogrammable elements of the FPGA fabric to produce the desired operational behavior. Because the FPGA fabric is reprogrammable, design changes are easily accomplished by changing the HDL, generating a new configuration file, and reprogramming the FPGA.

FPGA-based systems provide distinct advantages over processor-based and ASIC-based systems designed to perform identical tasks. Table 2.2 shows a comparison of the three hardware platform technologies and the advantages and disadvantages of each. Processor-based systems are preferable when cost is the overriding concern. ASICs are advantageous when performance is paramount. FPGA devices have the flexibility of a processor combined with the speed of a hardware implementation, providing high performance and fast development cycles. These devices provide an ideal environment for implementing high-performance network processing systems like the TCP-Processor.

FPGA devices are configured to determine the operation of each CLB and how logic, memory and I/O components are connected to each other. Once configured, the FPGA device exhibits the specified behavior until the device is either reprogrammed or reset. FPGAs are more configurable and less costly than ASIC-based implementations when dealing with small quantities of devices.

When compared with microprocessors, FPGA devices maintain flexibility, but also provide a significant increase in performance. This is due to the inherent parallelism in the FPGA fabric which can be exploited to achieve greater amounts of work completed per clock cycle. Additionally, FPGA devices lend themselves to pipelined logic operations which further increase the work performed per clock cycle. FPGA-based implementations

Table 2.2: Comparison of Hardware Processing Technologies

Hardware Technology	Pros	Cons
processor	ease of development short development cycle low development cost	slower performance
ASIC	high performance low power customized solution	high development cost long development cycle inflexible to changes
FPGA	high performance low power customized solution reconfigurable short development cycle product flxibility low development cost	not as fast or compact as ASIC

of the same algorithm, however, contain wired solutions which don't require stored programs. In addition, processors must read operands from memory and write results back to memory. FPGAs can be custom-coded to route the results of one operation directly to the input of a subsequent operation.

2.2 Challenges

There are many challenges associated with monitoring TCP flows within the context of high-speed networking environments. Packet sequences observed within the interior of the network can be different than packets received and processed at the connection endpoints. Along the way, packets can be fragmented, dropped, duplicated and re-ordered. In addition, multiple routes from source to destination may exist throughout the network. Depending on where a monitor is positioned within the network, it may only process a fraction of the packets associated with a flow being monitored. TCP stacks manage the sequencing and retransmission of data between network endpoints. Under some conditions, it is not possible to track the state of end systems perfectly and reconstruct byte sequences based on observed traffic [8]. This is due to inconsistencies in observed traffic with respect to the data actually processed at the end system. No single monitoring node can reconstruct

byte sequences if data packets take routes through the network which bypass the monitor. Even when all packets pertaining to a particular flow are routed through the monitor, there are still state transitions at an end system that cannot be fully known based on observed traffic. For example, suppose a monitoring application was attempting to reconstruct a two-way conversation occurring at an end application. By monitoring both inbound and outbound traffic for a particular flow, a close approximation of the conversation could be reconstructed, but there would be no guarantee that sequences of inbound and outbound data corresponded directly to the data sequences processed by the end application.

2.2.1 Performance

One of the main roadblocks for implementing a TCP flow monitor that operates on high-speed network traffic is the volume of work to be accomplished for every packet. The TCP-Processor's workload is driven by header processing. Larger packets have longer transmission times, which translates into having more processing cycles available in which to process the packet header. For minimum length packets, the TCP processor contends with a steady stream of back-to-back packet processing events. Figure 2.2 shows the number of clock cycles available for processing minimum length packets for various high-speed data rates. The arrow superimposed on the graph shows the technology trend for FPGA devices which corresponds to higher clock frequencies, greater numbers of reconfigurable logic blocks, and larger quantities of I/O pins. The operations required to perform flow classification and state management require approximately 40 clock cycles to process each packet. This processing time is dominated by the latency associated with accessing external memory.

Based on this performance, a single FPGA-based TCP processor operating at a clock frequency of 200MHz is capable of handling a steady stream of minimum length packets on an OC-48 (2.5Gbps) network. Because the system is not constrained by memory throughput, but by memory latency, performance can be doubled by instantiating parallel TCP processing engines. By saturating the bandwidth on the memory interface, the clock frequency required to process a steady stream of minimum length packets on an OC-48 link can be dropped to 100MHz. For 100 to 1500 byte packets, a single TCP-Processor circuit need only operate at 80MHz.

Figure 2.3 depicts the number of clock cycles available to process packets of different lengths at various network link speeds, given an operational clock frequency. Analysis of backbone connections within the interior of the Internet shows that the average packet

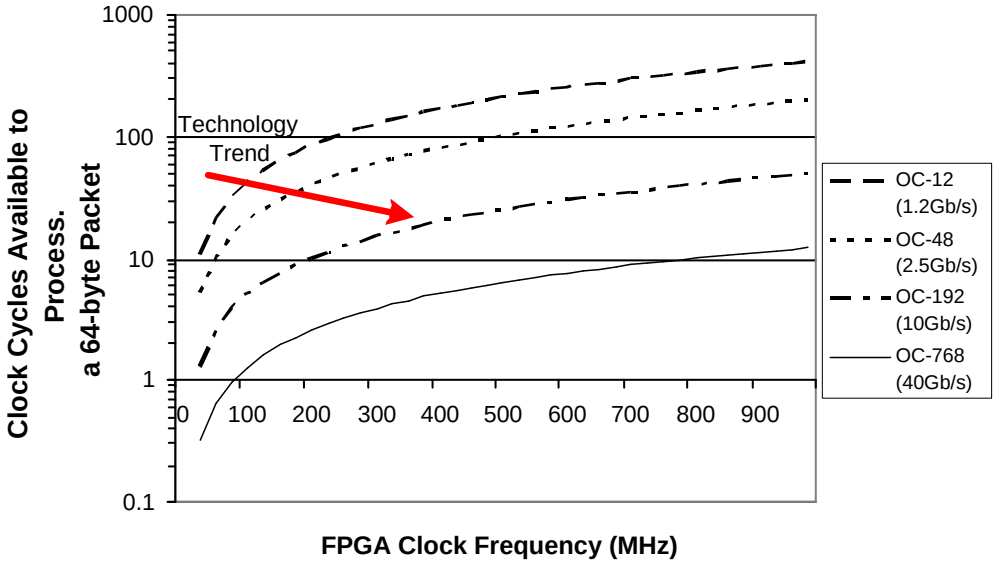


Figure 2.2: Clock Cycles Available to Process 64 byte Packet

sizes are approximately 300-400 bytes [127, 40]. Assuming again that 40 clock cycles are required to complete memory operations while processing each packet, a single TCP processing circuit operating at 200MHz can monitor traffic on a fully loaded OC-192 network link with average length packets. Including parallel TCP processing engines in the flow monitoring circuit can reduce this operating clock frequency to 100MHz.

2.2.2 Packet Classification

It is often, at least in the current Internet, easier to implement flow monitors at the endpoints of the network rather than at a monitoring node in the middle. The two main reasons for this are (1) only a small number of active connections need to be tracked at any one time and (2) the throughput is lower. However, neither of these assumptions hold true when the TCP processing occurs in a high-speed network monitoring environment. At any one time, there may be millions of active connections to monitor. Additionally, in order to process a worst-case scenario of back-to-back 40 byte packets on an OC-48 (2.5Gbps) link, a TCP processing circuit performs almost 8 million packet classifications per second.

A TCP connection is defined by the a 32-bit source IP address, a 32-bit destination IP address, a 16-bit source TCP port, and a 16-bit destination TCP port. A TCP processing system requires a 96-bit, four-tuple exact match using these fields in order to identify a TCP

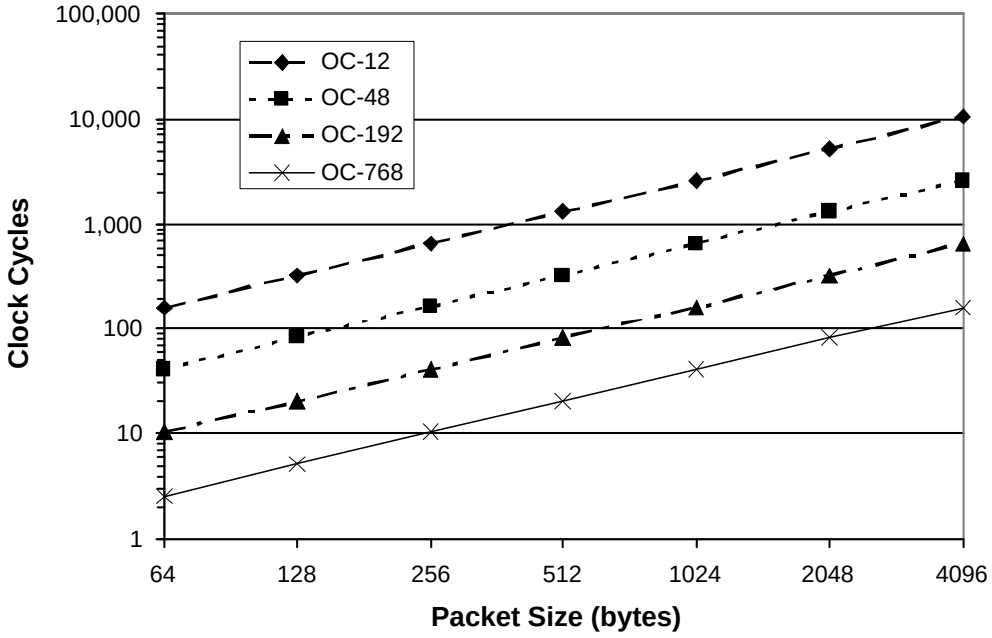


Figure 2.3: Clock Cycles Available to Process Packets of Various Sizes (200MHz Operation Frequency)

connection. The 8-bit protocol field of the IP header is usually included in this matching operation, but is not required for this circuit since this system performs classification operations only on TCP flows. In contrast, solutions to the general packet classification problem perform lookups based on a five tuple of source IP address, destination IP address, source port, destination port and protocol fields. Packet classification is a more difficult problem than exact matching and a large amount of research exists on enhancing packet classification algorithms in routers. These classifications can include a longest prefix match where rules containing more specific match criteria are selected over more general rules. An individual packet may match several entries, but only the results of the most specific match will be associated with the packet. In addition to longest prefix matching, packet classification employs priority schemes in which priorities assigned to overlapping rules resolve the lookup conflict. Additionally, header fields can be defined as ranges. This means that many packets can match a single rule during the classification process.

There are several distinct differences between general packet classification and the exact matching required for a TCP processing engine. First, since the matching occurs against an active network flow, there is no precomputation phase in which lookup tables

can be elegantly organized to reduce memory accesses associated with lookup operations. At startup, the exact matching effort has no information about future TCP connections which may appear in the network. This information has to be discovered at run time and precludes any optimizations which could be gained by precomputation.

Lookup optimization techniques such as rule expansion and table compaction [120, 31] are not applicable to the exact matching problem. These optimizations are targeted at performing longest matching prefix lookups by reducing the number of distinct prefix lengths that need to be searched. This is accomplished by expanding prefixes to small powers of two.

The complexity relating to the frequency and scope of lookup table changes for the exact matching of TCP packets for flow monitors far exceeds that of existing packet classification engines. Routers are subjected to route table updates at frequencies which require millisecond processing times. A single route change may result in a burst of route updates spreading through the network. The Border Gateway Protocol (BGP) supports the sending of multiple update messages to neighboring routers [54]. The vast majority of these route changes are pathological or redundant in nature and involve the removal and reinsertion of an identical route [68]. Route changes which only involve modification of the lookup result do not affect the size or shape of the lookup table itself and greatly simplify the update process. Conversely, an exact match lookup table for a TCP flow monitor is subjected to a table change whenever a new connection is established, or when an existing connection is terminated. A TCP processing circuit which processes data on a high-speed network link has to manage millions of active flows. In addition, there may be bursts of thousands, or tens of thousands of connections being established and torn down each second.

Sustained levels of TCP connections being created or destroyed can exceed normal traffic rates during Denial of Service (DoS) attacks and major events. DoS attacks occur when the network is bombarded with high volumes of illegitimate traffic. Superbowl half-time, natural disasters, and other global events can cause massive bursts in Internet traffic where large numbers of users access the Internet at the same time. Existing TCP flow classification and processing algorithms are not capable of handling the traffic loads which occur in these circumstances.

In order to handle this problem, hashing provides an efficient mechanism for storing N items in a table containing a total of M entries [79]. Hash functions are typically selected which result in an even distribution of entries across the whole table. Hashing techniques allow items to be accessed on average in $O(1)$ time. The drawback to hashing is that

performance degrades when two or more items map to the same entry, a condition known as a hash collision. Worst-case performance for hashing algorithms result in $O(M)$ time to access an entry. A perfect hashing technique based on static keys [42, 23] enables lookups to be completed with a worst-case of $O(1)$ memory accesses. This type of solution cannot be utilized by the TCP-Processor because the table entries are dynamic. Dietzfelbinger *et al*. describe a dynamic perfect hashing algorithm [30]. Unfortunately, this algorithm contains unrealistic space requirements and utilizes over 35 times the memory needed to store a set of data.

The TCP-Processor utilizes a variation of the open addressing hashing technique [63], originated by G. M. Amdahl in the 1950s [24]. This technique uses a direct indexed array lookup based on hash values to achieve the same worst-case of $O(1)$ memory accesses for a lookup. Hash buckets contain a fixed number of identically-sized entries. Burst memory operations which access sequential bytes of SDRAM devices are very efficient when compared to multiple accesses of individual memory locations. For this reason, the entire contents of the hash bucket can be accessed in a single memory operation. The exact match operation is performed while entries are clocked in from memory devices. This provides a single memory access flow classification solution.

2.2.3 Context Storage

A high-speed memory subsystem is required in order to maintain per-flow context information. At minimum, a 32-bit sequence number pertaining to the current position of the reassembly engine must be stored for each active flow. Realistically, keeping track of the TCP processing state of the end systems requires more context information. If the TCP flow monitoring service requires tracking data received by the end system, then it must also store the 32-bit acknowledgment number. Flow monitoring applications will likely require additional context information to be stored for each flow. Tracking both directions of a TCP flow requires additional context information for managing the other half of the connection.

Stateful inspection of network traffic was first introduced in the mid-1990s as a mechanism for providing better firewall and intrusion prevention capabilities [51]. The stateful inspection of TCP flows involves parsing protocol headers and tracking state transitions to ensure that the network packets are following proper TCP state transitions. For example, a data packet associated with a particular flow should never be received without first receiving TCP SYN packets initiating the connection. To perform stateful inspection

of TCP flows, the flow monitor needs to store additional context information for each flow so that information pertaining to the current state of a flow can be saved and retrieved when processing other packets associated with the same flow.

The need to update stored context information after processing each packet adds additional complexity to maintaining a per-flow state store. Under these circumstances, once packet classification has been completed, a read-modify-write memory access sequence will be issued. A delay occurs between the read and write memory operations when processing large packets to ensure that updates take place only after the complete packet has been processed. Updates to the flow context information should not occur if the TCP checksum for the packet fails and the checksum calculation does not complete until the last byte of data has been processed. This implies that the write cycle cannot start until after the complete packet has been processed. When dealing with back-to-back packets, the write operation of the previous packet can interfere with the lookup operation of the subsequent packet, which reduces throughput of the flow monitor. Additional delays can be caused by the memory devices themselves during memory refresh cycles and when requests are not distributed across memory banks. Without prior knowledge of packet sequences, it is impossible to design an algorithm which optimally distributes memory access across SDRAM devices.

2.2.4 Packet Resequencing

As noted earlier, TCP, a stream-oriented protocol, traverses the Internet on top of IP datagrams. Individual packets can be dropped or reordered while moving through the network toward their destination. The protocol stack at the end system is responsible for reassembling packets into their proper order, presenting the original data stream to the client application. Existing TCP stacks are tuned to process packets in their proper sequence and to degrade their performance when packets are received out of order [6].

TCP stream reassembly operations can require large amounts of memory. The TCP window size is negotiated by the endpoints at connection setup and has a maximum value of 65,535. The window scale factor can increase the window size by a factor of 2^{14}, which means the maximum possible window size is one GByte ($65535 * 2^{14} = 1$ billion) [57]. Given that the maximum window size is used for both directions of a TCP connection, an ideal TCP flow monitor which buffers packets for reassembly would require two Gigabyte

size buffers, one for handling traffic in each direction. A TCP flow monitor handling 1 million bidirectional flows would require two PetaBytes ($2 * 10^{15}$ bytes) of memory in order to store the state of all flows, which is not practical for implementations.

Recent analysis of backbone Internet traffic shows that only about 5% of packets generated by TCP connections are out of sequence [58]. With approximately 95% of TCP packets traversing the network in order, it is unlikely that worst-case patterns of retransmissions will ever be encountered. The arrival of packet sequences out of sequence requires packet buffering in order to passively track TCP stream content.

If all frames for a particular flow transit the network in order, the need for reassembly buffers can be eliminated. Ordered packet processing by the monitor can be guaranteed by actively dropping out-of-order packets. If it detects a missing packet, the TCP processor can drop subsequent packets until the missing packet is retransmitted. This methodology forces the TCP flow into a Go-Back-N retransmission policy where all data packets are retransmitted, thus eliminating the need for the receiving host to buffer and reassemble out-of-sequence packets [13].

When an end system receives an out-of-order packet, the action taken depends on which retransmission policy has been implemented. If the receiving system uses the Go-Back-N Automatic Repeat Request (ARQ), all out-of-order packets will be dropped and only packets in-order will only be processed. The transmitting system then resends all packets starting with the packet that was lost. Current implementations of the TCP stack utilize a sliding window ARQ and accept out-of-order packets within the window size negotiated at connection startup. Out-of-sequence packets are stored until the missing packets are received, at which time the buffered data is passed to the client application. Many operating systems, including Windows 98, FreeBSD 4.1, Linux 2.2 and Linux 2.4, still perform Go-Back-N ARQ [49].

Different extensible networking applications impose additional constraints on the TCP processing system. For instance, a security monitoring application may require that every byte of every flow that transits the monitoring device be properly analyzed. If a situation arises in which packets cannot be properly monitored due to resource exhaustion, packets should be dropped rather than passed through the device without analysis. In contrast, an extensible network application that monitors performance at an ISP may mandate that network throughput not be degraded during periods of duress.

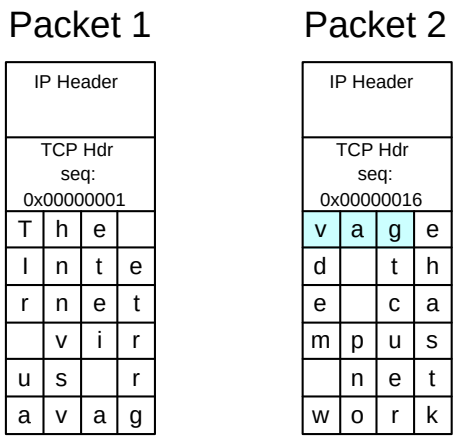


Figure 2.4: Overlapping Retransmission

2.2.5 Overlapping Retransmissions

During any TCP conversation, overlapping retransmissions can occur. While overlapping data is normal behavior for TCP, it can cause problems when performing flow reconstruction if not properly handled. In the example of overlapping retransmission illustrated in Figure 2.4, the character string *"The Internet virus ravaged the campus network"* traverses the network in two separate data packets. The first packet contains the 24 byte string *"The Internet virus ravag"* and the second packet contains the 24 byte string *"vaged the campus network"*. The three characters `"vag"` in the word *"ravaged"* are transmitted in both the first and the second packets. In order for a TCP flow monitoring service to operate correctly, it must be presented with an accurate reconstruction of the TCP data stream. When the second packet from the example is received, the shaded characters should not be processed by the monitoring application. To accomplish this task, the TCP processing engine must keep track of what data has been passed to the monitoring application and if overlapping retransmissions occur, needs to indicate which data in each packet should be processed and which data should be ignored.

2.2.6 Idle Flows

TCP supports the notion of an idle flow. An idle flow is a valid TCP connection between two end systems where the connection is idle and no packets are transmitted or received. The TCP specification places no limits on the duration of a silent period. The idle period

can last minutes, hours, or even days. For network monitors, an idle flow occupies monitoring resources even though there is no traffic. In addition, it is impossible for the network monitor to differentiate between a flow which is idle and a flow which has been abnormally terminated for reasons of computers crashing, a network or power outage, or a change in routing tables which sends packets along a different path through the network. In all of these cases, the network monitor must apply a heuristic to manage these flows.

A TCP flow monitor can handle idle flows in several ways. One way is to maintain state until observance of a proper shutdown sequence. If shutdown sequences are not processed by the TCP flow monitor, then resource reclamation will not occur which will in turn lead to the eventual exhaustion of all the resources used to track network flows. Another way to handle idle flows is to have the TCP flow monitor drop all packets associated with flows that cannot be monitored due to unavailable resources. This guarantees that all data passing through the monitoring station will be monitored, but has the potential to eventually shut down the network by not allowing any packets to pass in a situation where all resources are consumed with improperly terminated flows. A third option would be to implement an aging algorithm with a timer. A flow is removed from the monitor after a predefined period of idle time. A network monitor which employs this method, however, can easily be circumvented if the end user alternately sends pieces of a message and then waits for a long idle period before transmitting another segment. A fourth method is to reclaim resources when all available resources are utilized. The process of reclaiming resources can induce delays during normal operation due to the extra memory operations required to reclaim the resources. These additional memory operations induce delays in the system and can, in turn, cause other resource utilization problems.

2.2.7 Resource Exhaustion

Stream reassembly services constitute a key component of the TCP processing engine. Utilizing buffers to store and re-order network data packets requires a strategy to cope with depleted buffers. A time-ordered list of packet buffers is maintained by the packet storage manager. If all of the packet storage resources are utilized, the oldest entry in the list is reassigned to store the new packet data.

The critical nature of resource exhaustion and its likelihood demands clarification. Several resources associated with a high performance TCP flow monitor have the potential to be over-utilized. These include the flow classifier, the context storage subsystem,

the stream packet buffers used in stream reassembly, the computing resources, and other internal buffers and FIFOs.

High throughput rates should be maintained during worst-case traffic loads. As previously noted, 100% utilization of the context storage subsystem can result from idle flows and improperly terminated flows. Timer-based maintenance routines which periodically sweep these tables to free up idle resources can be employed to reduce the probability that resources will be exhausted. However, during periods of exceptionally high connection volume, context storage resources can still be oversubscribed. In addition, some maintenance operations require exclusive access to memory resources while performing reclamation operations. This can cause memory contention issues and adversely affect throughput. An alternative approach is to maintain a linked list of least-recently-used flows. When all resources are utilized, the flow at the bottom of this list is reclaimed so that the new flow can be processed. Maintaining this link list structure adds the additional overhead of performing six memory operations for each packet processed. As shown in Figure 2.2 on page 20, there are a limited number of clock cycles available in which to process a small packet. The TCP-Processor handles cases of the exhaustion of flow classification resources with a least-recently-used recovery mechanism.

Applications which process the data streams provided by a TCP processing service require the ability to store context information for each flow. After processing the payload section of an individual network packet, the current state of the application processing subsystem has to be stored for later use. In addition, prior to processing data from a packet, context information has to be retrieved and loaded into the application. This implies that the application requires a memory read operation at the start of every packet and a memory write operation at the end of every packet. The size of this context information is dependant on the specific requirements of the application. In order to maintain processing throughput, the total amount of per-flow data should be minimized. This limitation also applies to the per-flow context information maintained by the TCP processing engine.

2.2.8 Selective Flow Monitoring

A TCP flow monitor which reconstructs byte streams for application processing should be flexible and provide support for multiple service levels which can be assigned and altered on a per-flow basis. Different types of services require different mechanisms to process flows. For high volumes of traffic, flow monitoring may only be needed for a subset of the

total traffic. An ISP, for example, may want to offer different value-added TCP flow processing services to different clients. To do this, a TCP flow monitor must perform different operations for different flows. For example, connections originating from a particular subnet may be configured to bypass the monitoring subsystem altogether. Other flows may be subjected to best effort monitoring while a third subset of the traffic requires complete monitoring. In this context, complete monitoring means that packets are not allowed to pass through the monitor without first being scanned by the monitoring service. An additional constraint is that the monitoring requirements for a flow may change during the lifetime of that flow.

2.2.9 Multi-Node Monitor Coordination

Gateway routers act as a single entry point for smaller isolated networks connecting into a larger mesh network, and the TCP flow monitors discussed until now have been placed at this point. A TCP processing engine placed at a gateway router is guaranteed to see all packets associated with any communications between machines on either side of the router. In contrast, core routers are usually connected in mesh-like patterns consisting of multiple traffic paths through the core network. To illustrate this distinction, Figure 2.5 shows an example network with both core and gateway routers. Two end systems identified as A and B are connected to gateway routers at different locations in the network. Communication between nodes A and B can travel over two different paths through the network as indicated by the thick router connections.

Placed at core routers, TCP flow monitors may receive some, but not all, of the total traffic for an individual flow. A robust TCP monitoring service must coordinate monitoring across multiple core routers. Through coordination, multiple flow monitors can work together to provide comprehensive monitoring of traffic. If packets associated with a particular flow are detected bypassing the monitoring node by another core router, they could be rerouted to the appropriate monitoring station. A challenge to coordinating multiple flow monitors is that excessive amounts of traffic can be generated. In addition, route changes by network administrators, fiber cuts, and device failures lead to dynamic changes in the normal flow of traffic.

2.2.10 Fragmentation

Packet fragmentation, the splitting of a larger packet into several smaller packets, occurs below the TCP layer at the IP layer. Fragmented packets need to be reassembled before the

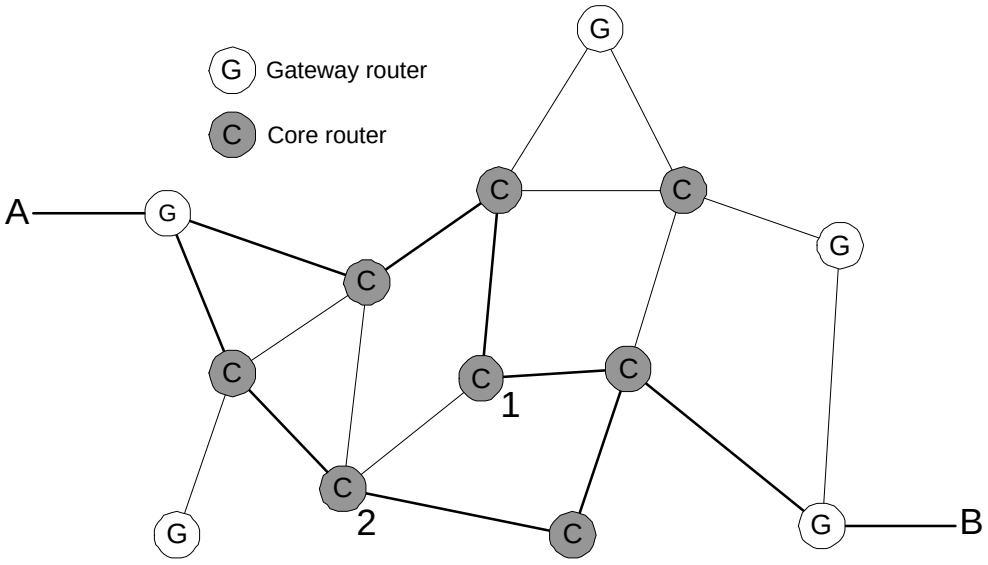


Figure 2.5: Multi-Node Monitor

processing of the TCP packet can be completed. Similar to the Packet Resequencing problem (page 24), IP packet fragments can be reordered or dropped from the network, complicating the task of defragmentation. The IP header in each fragment contains information about how to reassemble the fragments into the original packet. Packet defragmentation normally occurs prior to entering the TCP processing subsystem.

There are three possible methods of dealing with fragmented packets. The first would be to have the IP processing subsystem perform packet defragmentation services. Since the TCP flow monitor only receives fully formed TCP packets, no changes would be necessary. A pool of temporary buffers would hold fragments. When all of the individual IP fragments had been received, the IP processing engine can then rebuild the original packet and pass it along to the TCP processing layer. This method of performing packet defragmentation at the IP layer is almost identical to that associated with packet resequencing at the TCP layer.

The second method of dealing with fragmented packets would be to enhance the TCP-Processor to deal with multiple IP fragments containing a single TCP packet. The TCP processing engine would still require that the fragments be processed in order, but would not require that they be reassembled. This complicates the flow classification process because a lookup would have to be performed for each of the fragments based on the IP source address, IP destination address, and IP identification fields. In addition, all fragments would have to be held until the last fragment is received so that the TCP checksum

can be updated. Only then could the data contained within the fragments be passed along to the monitoring application.

The third method would be to develop a slow path which is reserved for handling low occurrence events. Algorithms which process network packets are frequently optimized so that the majority of packets are processed in a highly efficient manner. The term slow path refers to the more time consuming sequence of operations required to process low occurrence events, such as packet fragments. This slow path would utilize a microprocessor and/or additional logic along with SDRAM memory to perform packet defragmentation. Once all of the fragments have been reassembled, the resultant packet would then be re-injected into the fast path for normal processing. Since the slow path would only be utilized for exception processing, it could have a much lower throughput capacity.

Processing efficiencies can be gained when packet storage capabilities are shared between the packet defragmenting logic and the TCP packet resynquencing logic. The second method for handling fragmented packets describes a scenario where packet defragmentation is handled by the TCP processing module, which locates resequencing logic and defragmentation logic in the same module. This design supports the development of a single packet storage manager capable of storing and retrieving arbitrary network packets.

2.2.11 Flow Modification

Extensible networking services, such as those which prevent the spread of Internet-based worms and viruses, require advanced features which support the modification of flows. In particular, intrusion prevention services require the ability to terminate a connection, block the movement of data associated within a particular flow for a period of time, or modify the content of a flow in order to remove or disarm a virus. To support the blocking and unblocking of flows, a protocol processing engine must maintain enough per-flow context information to determine whether or not the flow should be blocked and at which point in the data stream traffic should be blocked. To support flow termination, a TCP FIN packet should be generated. In addition, state information indicating that the flow is in a termination state must be stored in case the generated TCP FIN packet is dropped at some later point in the network. In order to handle this case properly, the processing engine should enter a state where it drops all packets associated with the flow and regenerates TCP FIN packets until the connection is properly terminated.

Supporting the modification of flows adds additional levels of complexity to the TCP circuit. Two classes of flow modification are considered. The first involves altering content

in data packets associated with a flow. In this case, since no data is inserted or deleted (i.e., the number of bytes traversing the network is the same), the modification can be supported by processing traffic in a single direction. Additional per-flow state information is needed in order to store the new data bytes along with the sequence number at which the replacement took place. This information is vital to allow identical byte replacements to be performed for any retransmitted packets that include altered sections of the data stream.

In order to support flow modifications where data is either added to or removed from a TCP data stream, packets must be processed in both directions of the TCP connection. The reason for this is that acknowledgment sequence numbers contained in the return packets must be modified to account for the data that was either inserted or removed from the stream. The TCP flow monitor must also track acknowledgment packets and generate TCP packets containing data which was inserted into the stream in accordance with the TCP specification [53]. In addition, packets may need to be broken into several smaller packets if the act of data insertion causes the packet to exceed the MTU for the flow. As with the previous case, the per-flow context information should be expanded in order to store the position of the flow modification, the number of bytes inserted or deleted, and a copy of the data if an insertion has taken place. This information can be removed from the context storage area once an acknowledgment is received for the modified portion of the flow which indicates that the data has been received by the end system.

Another potential flow modification service alters response traffic based on content observed in a preceding request. This feature is useful when filtering web requests. A TCP flow monitor searches web requests for specific, predefined content. When the content is detected, a properly formatted HTTP response is generated containing an appropriate response. The request packet is then dropped from the network, preventing the request from ever reaching the web server. Finally, the TCP connection is terminated to end the session. Corporate firewalls can employ this technology to grant employees access to Internet search engines, while limiting their searching capabilities.

2.2.12 Bi-Directional Traffic Monitoring

For certain content scanning and filtering services, monitoring outbound traffic independently of inbound traffic is acceptable. The TCP flow monitor may need to reconstruct a full TCP conversation for analysis which requires the processing of both directions of a connection. Additionally, these services may want to process acknowledgment messages in order to determine what data was actually received and processed by the end system. In

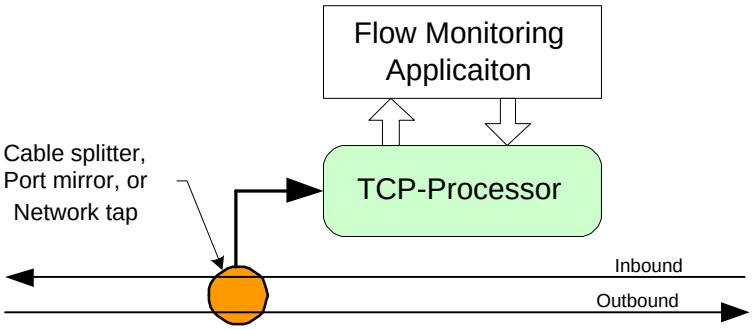


Figure 2.6: Bi-directional Flow Monitoring

order to provide this capability, the TCP processing engine needs to process both directions of a TCP connection utilizing a unified flow context block.

The TCP-Processor supports coordinated monitoring of bi-directional traffic by using a hashing algorithm that generates the same result for packets traversing the network in both the forward and reverse directions. The state store manager stores inbound and outbound context records in adjacent memory locations. The flow identifiers assigned to the inbound and outbound packets pertaining to the same TCP connection differ only by the low bit. This allows the monitoring application to differentiate between the inbound and outbound traffic while performing coordinated flow monitoring. In order to monitor traffic in this manner, data traversing the network in both directions must be passed to the TCP-Processor. Figure 2.6 shows a passive monitoring configuration where bi-directional traffic is monitored with the aid of a content duplicating device, such as a network switch configured to do port mirroring.

Asymmetric routing can occur when data traverses through core routers. With asymmetric routing, outbound traffic takes a different path through the network than inbound traffic associated with the same connection. Hot potato routing can cause this type of behavior and is occasionally used by carriers wanting to limit the amount of transient traffic they carry. In this instance, when packets are detected which neither originate nor terminate within the carrier's network, the packets are routed to a different network as soon as possible. A TCP flow monitor capable of analyzing bi-directional traffic which has an asymmetric data path through the network must employ additional logic to coordinate events amongst multiple monitoring nodes in order to ensure that the conversation is properly monitored.

Chapter 3

Related Work

This chapter reviews high-speed protocol processing efforts which are similar or related to this research. It provides a brief overview of network monitoring systems, along with examples of both software-based and hardware-based network monitors and their current limitations. It describes various packet classification techniques and discusses related technologies which perform TCP flow processing. Finally, it presents related work in hardware-accelerated content scanners, as these applications are ideal candidates for integration with the TCP-Processor technology.

3.1 Network Monitoring Systems

There are many different types of network monitors and protocol analyzers available today. These network monitors have a wide range of features and capabilities. Understanding the differences between the various monitors is challenging because terminology is often vague or misused. IP-based networks are commonly referred to as TCP/IP networks, although TCP and IP are two separate protocols which perform different functions in a network. Transmission Control Protocol (TCP) [53] is a stream-oriented protocol which maintains a virtual bit pipe between end systems. Internet Protocol (IP) [52] is an unreliable datagram delivery service which provides packet routing between two end systems.

Figure 3.1 shows a taxonomy of network monitors. The proposed TCP monitor performs full flow analysis as indicated by the gray oval in the figure. TCP stream reassembly operations are required to fully process TCP flows, in addition to IP and TCP header processing. This operation separates TCP flow monitors like the TCP-Processor from other types of network monitors. For example, the `tcpdump` [83] program performs

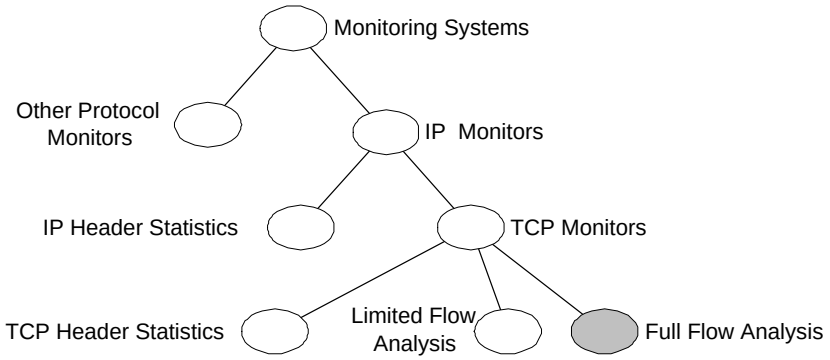


Figure 3.1: Taxonomy of Network Monitors

packet capturing and filtering services, but does not provide a full range of TCP monitoring features. `Tcpdump` maintains a minimal amount of per-flow state information in order to display relative sequence numbers instead of absolute sequence numbers. Stream reassembly and stateful inspection features, however, are not included in the utility.

The TCP-Processor takes a novel approach to network packet processing by providing full TCP flow analysis operating at multi-gigabit per second data rates while managing millions of active connections. In addition, the TCP-Processor operates as an in-band data processing unit, which enables active processing on network flows. This separates the TCP-Processor from other types of packet processors which do not have the performance or capacity to perform TCP stream processing on large numbers of flows in high-speed networking environments.

3.2 Software-Based Network Monitors

A multitude of software-based packet capturing and network monitoring applications exist today. These monitors offer a wide range of capabilities, but because they are software-based, most have performance limitations that prevent them from effectively monitoring high-speed networks. In order to improve performance, many software-based monitors only inspect packet headers, ignoring the data payload.

Packet capturing tools such as `tcpdump` [83], Internet Protocol Scanning Engine [46], and Ethereal [95] support the capture and storage of network packets. These types of tools receive copies of network traffic by interfacing directly with lower layer networking drivers. These applications work well for monitoring data at low bandwidth rates, but

because they execute in software, their capabilities are limited. Performance of these monitors can be improved by limiting the number of bytes per captured packet. This tradeoff is acceptable when examining only protocol header fields. TCP data stream analysis can be performed as a post-processing operation if all data bytes associated with a particular flow are captured.

`HTTPDUMP` is an extended version of `tcpdump` which supports the analysis of web-based HyperText Transfer Protocol (HTTP) traffic [105]. The extra filtering logic required for processing the HTTP protocol reduces this tool's performance. Since a single type of TCP traffic is targeted, `HTTPDUMP` is unable to operate as a general purpose flow monitor.

`PacketScope` [2] is a network traffic monitor developed at AT&T in order to provide for the passive monitoring of T3 (45Mbps) backbone links. This monitor utilizes a network tap which extracts a duplicate copy of the network traffic. Traffic is then routed to the `tcpdump` application. The first 128 bytes of each packet are captured and stored to support analysis. `BLT` [37] leverages the `PacketScope` monitor to perform HTTP monitoring. It decodes the HTTP protocol along with packet headers and generates a log file which summarizes the activity of these flows. PacketScope does not guarantee the processing of all packets on the network. Instead, results are obtained by processing only the majority of HTTP packets.

The Cluster-based Online Monitoring System [80] supports higher data rate HTTP traffic monitoring. Even though this is also a software-based network traffic monitor, it achieves higher throughput rates by utilizing a cluster of machines to analyze the network flows. With eight analysis engines operating, however, traffic is not consistently monitored at a full 100Mbps line rate. Software-based flow monitors are clearly unable to monitor large numbers of flows at multi-gigabit per second line rates.

All of these software-based network monitors have problems monitoring traffic flowing at a 100Mbps data rate. There are several reasons for this performance limitation. Microprocessors operate on a sequential stream of instructions. This processing model dictates that only one operation can occur at a time and many instructions may be required to perform a single operation. Processing a single packet may require executing thousands or tens of thousands of instructions. Software-based monitors with greater complexity require more instructions, resulting in larger code sizes. Larger code sizes increase the likelihood that instruction caches will have to be flushed and reloaded during the processing of a packet. Software-based network monitors also require several data copies, which degrades the overall throughput of the network monitor. For example, data first must be copied from

the hardware into main processor memory. Then, an additional data copy may be made during processing of the various protocol layers. Finally, the data must be copied from kernel memory into an end user's application data buffer. Application signalling, system calls and interrupt processing expand the amount of processing overhead which further reduces the throughput of the monitor.

To overcome the limitations associated with software-based packet processing systems, the TCP-Processor exploits pipelining and parallelism within a hardware-based processing environment to achieve much higher levels of performance. Because there is no operating system involved, the overhead associated with system calls, scheduling, and interrupt processing is eliminated. Additionally, the TCP-Processor does not store network packets in external memory prior to or during normal packet processing. This eliminates processing delays due to memory latency, memory bandwidth, and memory management. Furthermore, much like the assembly line approach to building automobiles, the TCP-Processor employs pipelining to increase the throughput of the system. The various components of the TCP-Processor can each operate on a different network packet at the same time. The TCP-Processor also employs parallel processing techniques which allow separate logic blocks to perform processing on identical portions of a network packet at the same time. These design choices combine to enable the TCP-Processor to process TCP data streams at over 2.5 Gbps.

3.3 Hardware-Based Network Monitors

The performance advantages associated with a hardware-based TCP monitor close the gap between the transmission speeds of high-speed networks and the silicon scaling predicted by Moore's Law (see Figure 2.1 on page 14). Necker *et al*. developed a TCP-Stream Reassembly and State Tracking circuit that is capable of analyzing a single TCP flow while processing data at 3.2Gbps [91]. The circuit was tested utilizing an FPGA device which tracks the state of a TCP connection and performs limited buffer reassembly. The current implementation of the circuit only processes a one-way traffic for a single TCP connection. The authors outline the components which need to be replicated in order to manage multiple flows. They suggest instantiating a separate copy of the *Connection-State-Machine* for each reassembled TCP stream and sharing the logic of the *Ack/Seq Tracking Unit* by storing connection specific data in multiplexed registers. Based on resource utilization, the authors estimate 30 reassembly units can fit within a single Xilinx Virtex XCV812E FPGA. There are inherent difficulties associated when moving from a single flow processor to a

multi-flow processor. Signal fan-out delays, context storage and retrieval delays, and limited amounts of logic, memory and interconnect resources prohibit hardware-based flow processors to be scaled in this manner above a few hundred concurrent flows.

Li *et al*. outline an alternative approach to FPGA-based TCP processing [72]. Their design provides TCP connection state processing and TCP stream reassembly functions for both client and server-directed traffic on a single TCP connection. It maintains a 1024 byte reassembly buffer for both client and server side traffic and drops packets lying outside of the reassembly buffer space on receipt are dropped. Portions of the circuit design have been implemented and process data at 3.06 Gbps. Each individual TCP connection requires a separate instance of this circuit. A maximum of eight TCP flows could be monitored using a Xilinx Virtex 1000E FPGA. This type of system does not scale and is insufficient for monitoring the large number of flows found on high-speed network links.

Unlike other FPGA-based TCP processors, the TCP-Processor was designed from the ground up with the notion that it must support large numbers of simultaneous TCP connections while operating at multi-gigabit data rates. The problems associated with context storage and retrieval while operating in this environment were addressed from the start. To accomplish this task, the internal structure of the TCP processing engine localizes packet processing which requires persistent state information and optimizes the design so this information can be easily stored and reloaded with the processing of each network packet. A separate state store manager component manages operations to an external memory device which stores per-flow context information and completes the flow classification operation. These techniques enable the TCP-Processor to process millions of simultaneous TCP connections while processing data at multi-gigabit data rates.

3.4 Packet Classification

A large amount of effort has been applied to date to the problem of performing high-speed packet classification [82, 98, 26, 119, 135, 126, 115, 100, 29]. Until the mid-1990s, most work centered around performing a longest prefix match on a 32-bit destination IP address field. Network equipment performing IP packet forwarding requires this type of operation. More recent research examines the use of multiple header fields to perform packet classification. The various classification techniques can be broken into four separate groups: CAM-based algorithms, trie-based algorithms, multidimensional or cutting algorithms and other research which doesn't fit into one of the other categories. The algorithms for performing high-speed packet classification in each of these categories are summarized below.

Content Addressable Memories (CAMs) provide a fast mechanism for performing lookups which result from parallel comparison operations [82]. These devices perform binary comparisons that require large table expansions when wildcards are included in the search field. Because CAM devices contain more support circuitry than SRAM devices, they are more expensive, slower, have lower densities, and require more power than commodity memory devices. Ternary CAMs (TCAMs) provide an extension to the basic CAM device by storing three possible states (0, 1, X-don't care) for each bit. This feature is typically implemented with a comparison value and a mask value. Since two values are stored for each entry, TCAMs have twice the memory requirements of regular binary CAM devices. This increases the cost and power consumption for these devices. Shah and Gupta [111] introduced two algorithms, prefix length ordering and chain ancestor ordering, to reduce the number of memory operations required to perform incremental updates to TCAM devices. Panigrahy and Sharma proposed paging and partitioning techniques in order to improve throughput and reduce power consumption in TCAM devices [98].

Trie-based packet classification algorithms are based on tree structures in which a single node is associated with every common prefix and exact matches are stored as leaves under that node. Trie-type retrieval algorithms originated in the late 1960s [90]. Most notably, the BSD operating system employed a trie lookup mechanism in order to perform next hop packet routing [60]. Enhancements to the basic trie algorithm used for IP forwarding proposed by Degermark *et al*. can improve the lookup performance to 1 million IP packet classifications per second [26]. In addition, the *grid-of-tries* concept by Srinivasan *et al*. extends the normal binary trie to support lookups on two fields [119].

Waldvogel *et al*. developed a high-speed prefix matching algorithm for router forwarding which can process 10-13 million packets per second [135]. This algorithm introduces mutating binary trees with markers and pre-computation in order to bound worst-case lookup performance to logarithmic time. While extremely efficient when performing lookup operations, this algorithm does not easily support incremental updates. The Fast Internet Protocol Lookup (FIPL) architecture [125, 126] guarantees worst-case performance of over 9 million lookups per second. FIPL maintains lookup performance while simultaneously handling 1,000 route updates per second. It is implemented in FPGA logic utilizing one external SRAM module and operates at 100MHz. For testing, 16 thousand route entries, requiring only 10 bytes of storage for each entry, were loaded into the FIPL engine.

More recently, complex routing instructions require routers to perform packet classification on multiple fields. This usually involves the 5-tuple of source address, destination

address, source port, destination port, and protocol fields. The Recursive Flow Classification (RFC) algorithm exploits structure and redundancy found in rule sets in order to improve performance [47]. By mapping rule sets into a multi-dimensional space, a technique of performing parallel range matches in each dimension can improve classification speeds to a million packet classifications per second [69]. The Aggregated Bit Vector (ABV) scheme extends this algorithm by adding rearrangement and recursive aggregation which compress tables and reduce the number of memory accesses required to perform a lookup [3]. Gupta and McKeown describe a hierarchical cutting technique [48] which utilizes rule preprocessing to build a tree whose nodes represent a subset of the rule sets. HyperCuts extends this concept to allow simultaneous cutting in multiple dimensions and reduces memory requirements by pulling common rules up to higher nodes within the tree [115].

`SWITCHGEN` is a tool which transforms packet classification rules into reconfigurable hardware-based circuit designs [59]. The goal of this approach is to achieve 100 million packet classifications per second, or sufficient throughput to manage traffic on an OC-768 data link. Prakash *et al*. propose a packet classifier which performs lookups utilizing a series of pipelined SRAMs [100]. One billion packet classification lookups per second could be supported with this technology.

Most recently, Dharmapurikar *et al*. describe a Bloom filter hashing technique to perform packet classifications [29]. A Bloom filter involves hashing an input several times using different hashing functions. The resultant values are used as direct memory indexes into sparsely populated hash tables. These hash table lookups can be performed in a single clock cycle utilizing on-chip memories. If all of the values from the lookups return true, then there is a high probability that a match has been found. The Bloom filter technique is subject to false positive matches which require an additional comparison in order to validate the result, but a false negative will never be returned. It is capable of performing an average of 300 million lookups per second, with worst-case performance of 100 million lookups per second. By comparison, TCAMs operate at 100 million lookups per second, consume 150 times more power, and cost 30 times more than the commodity SRAM devices that can be utilized in a packet classifier based on Bloom filter technology.

The TCP-Processor currently uses a modified open addressing hash scheme for managing per-flow context information. This modification involves limiting the length of hash chains in order to provide deterministic behavior when performing flow classification and retrieving per-flow context information. While excessive hash collisions can lead

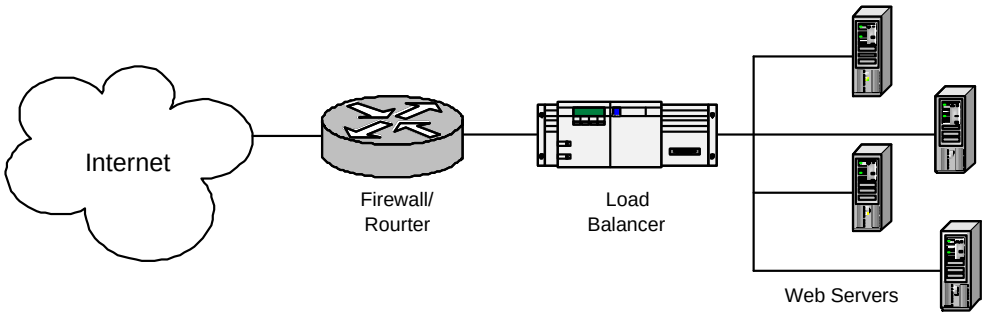


Figure 3.2: Web Farm with Load Balancer

to the incomplete monitoring of a TCP flow, the occurrence of hash collisions can be reduced by selecting hash algorithms which evenly distribute flows throughout memory and appropriately sizing memory for the number of expected flows. In addition, flow classification operations are localized within the TCP-Processor so that new flow classification algorithms can be easily incorporated without requiring major changes to the system.

3.5 Related Technologies

The technology employed by load balancers, SSL accelerators, intrusion detection systems, and TCP offload engines is different than the TCP flow processing architecture described in this dissertation. These devices do contain components that process the TCP protocol, and therefore have related processing requirements. This section discusses the operation of these devices and compares their TCP processing features to this research.

3.5.1 Load Balancers

Load balancing devices distribute the request load of a common resource among *N* number of servers. They are useful in web farms where request loads exceed the capacity of a single machine. Another advantage of load balancing devices is that they provide failover support, masking the failure of a single backend server by routing requests that would otherwise have failed to other functioning servers. In web farms, the load balancing device is typically placed between a firewall router and a group of web servers. The firewall router provides access to the Internet and the web servers provide content to the Internet community. Figure 3.2 illustrates a typical web farm.

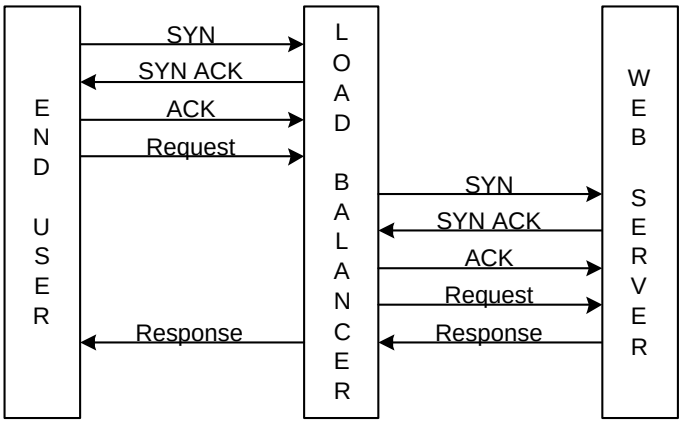


Figure 3.3: Delayed Binding Technique

One of the features of load balancing devices is content-based (or cookie-based) routing. In a content-based routing scenario, the load balancer directs requests to a specific server based on information contained within the request. Since a connection has to be established before content can be delivered, load balancing devices employ a concept called delayed binding. It allows the load balancing device to complete the TCP connection and receive the web request. On receipt, the request can be parsed in order to determine which of the backend servers should receive the request. The load balancer then initiates a TCP connection with the appropriate web server and forwards the request. When the web server generates a response, the load balancer adjusts the TCP header fields of the response packets and returns the response to the end users. Figure 3.3 illustrates the TCP interaction between the end users, the load balancer and the web server. Many commercial vendors of load balancing equipment employ delayed binding [92, 35, 18, 39, 101].

A load balancing processes TCP header fields when performing content-based routing. The main differences between load balancing devices and the research presented in this dissertation are:

- Load balancers are in close proximity to a web server or end system.
- Load balancers partially terminate the TCP connection.
- Load balancers only scan the initial part of the TCP flow.
- Load balancers only scan traffic in one direction.
- Load balancers are limited in the number of simultaneously flows that they can monitor.
- The amount of traffic that current generation load balancers can handle is limited.

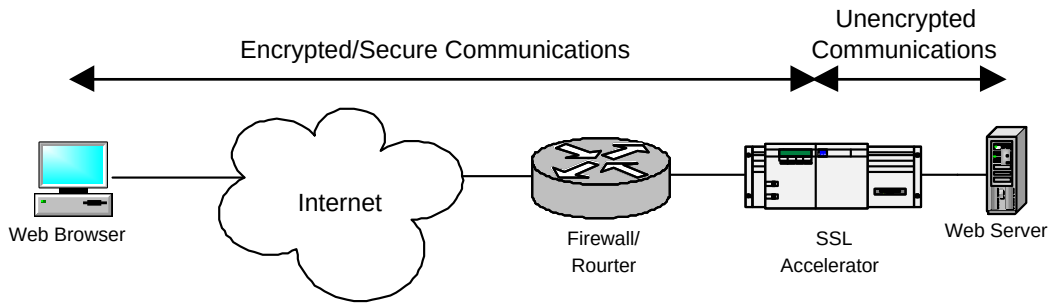


Figure 3.4: SSL Accelerator Employed at Web Server

In contrast to load balancers, the TCP-Processor is designed to perform full TCP flow analysis for millions of flows at multi-gigabit data rates. In addition, there is no requirement that the TCP-Processor be placed next to an end system and it is also capable of processing bi-directional traffic. Load balancers fulfill a specific need for Web Servers and are not capable of performing the functions of the TCP-Processor.

3.5.2 SSL Accelerators

Secure Sockets Layer (SSL) is a protocol for providing an encrypted and secure data path between two endpoints [43]. Web browsers employ this protocol to support secure communication with a Web server, preventing eavesdropping, data tampering, and data forgery.

The processing overhead required to implement the SSL protocol demands SSL acceleration. When a web server processes SSL, its encryption/decryption algorithms consume much of the server's processing power. Several network vendors offer SSL accelerators as a solution to this problem [93, 36, 20, 84]. An SSL accelerator is an external device which implements the SSL protocol, including key management, encryption, and decryption processing. An SSL accelerator frees up the web server or end processing system from having to perform the compute-intensive operations required to provide a secure communications environment. Figure 3.4 shows the placement of an SSL accelerator in the network. The SSL accelerator is tightly coupled with one or more web servers and provides secure communication services for TCP connections at a single location.

An SSL acceleration device performs protocol processing on TCP connections. The main differences between SSL acelerators and the research presented in this dissertation are:

- SSL accelerators are in close proximity to a web server or end system.

- SSL accelerators terminate and re-establish TCP connections.
- SSL accelerators act as protocol translators, communicating via secure connections on one side of the device and via normal non-secure connections on the other side.
- The number of simultaneous flows SSL accelerators can process is limited.
- The amount of traffic that current generation SSL accelerators can handle is limited.

SSL accelerators terminate and re-establish a TCP connection with a back-end Web Sever. They are designed to be placed in very close proximity to the Web Servers to which they connect. The TCP-Processor does not terminate and re-establish TCP connections and has flexible placement requirements. It is designed to perform full TCP flow analysis for millions of flows a multi-gigabit data rates. SSL accelerators are designed for encyrption/decryption services and not general TCP flow processing like the TCP-Processor.

3.5.3 Intrusion Detection Systems

Most intrusion detection systems in use today have software-based implementations. Snort [104] and other software-based intrusion detection systems have performance limitations which prevent them from processing large quantities of traffic at high data rates. Software-based intrusion detection systems contain performance limitations which prevent them from being able to fully process traffic on a heavily loaded 100Mbps network link [107]. In order to improve performance, rule sets are tailored to reduce the amount of required text searching. In addition, instead of performing TCP processing and stream reassembly operations to accurately interrogate stream payloads, Snort, for example, is typically configured to inspect only payloads on an individual packet basis. Virus scanners such as *VirusScan* [81] and *AntiVirus* [123] operate on end systems where they monitor data as it passes between the application and the TCP stack.

Li *et al*. have implemented portions of an intrusion detection system in reconfigurable hardware [71]. Header processing and content matching logic utilize FPGA-based CAM circuits to search for predefined content in network packets. The content matcher searches for a single signature based on the result of the header processing operation. Data is clocked into a 160 bit wide sliding window matching buffer 32 bits at a time. This supports the matching of digital signatures of up to 16 $((160 - 32)/8)$ characters in length. Although the circuit can process data at 2.68Gbps, there is no support for managing TCP flows.

Granidt (Gigabit Rate Network Intrusion Detection Technology) combines a hardware component and a software component to create an intrusion detection system [45].

The hardware component utilizes multiple FPGA devices to perform header processing and content matching using logic-based CAMs. A 32-bit PCI interface connecting the FPGA devices to the main computer limits processing bandwidth to 696Mbps. The hardware portion of the design can process data at 2Gbps. The Granidt system does not perform TCP flow processing and only searches for content on a packet by packet basis.

Of the available commercial intrusion detection systems [55, 19, 38], none are capable of processing millions of data flows at multi-gigabit line rates. Accomplishing this goal requires high performance TCP processing, flow classification, stream reassembly, and content scanning techniques. Vendors apply heuristics which limit the stream reassembly or content scanning operations in order to achieve higher throughput rates. Most of these devices support lower traffic throughput rates (on the order of a few hundred Mbps) and have limited capacity for supporting large numbers of simultaneous connections. The target placement for these devices is typically within client networks where data rates and the total number of connections are limited.

State-based processing languages aid in the definition and tracking of state transitions associated with network attacks [99, 64, 104]. Kruegel *et al* developed a stateful intrusion detection system for use in high-speed networks which divides traffic on high-speed links and disperses the processing load to an array of computers running software-based analyzers [64]. This system employs an extensible state transition-based language (STATL) and a state transition framework (STAT) which simplify the process of describing and detecting network-based attacks [32, 134]. It also employs a static configuration for splitting traffic amongst the monitoring systems and therefore cannot react to changes in network traffic patterns. This can lead to overload conditions on individual flow monitors. A single computer running STAT is capable of processing data at 200 Mbps.

WebSTAT also leverages the STAT framework to monitor HTTP requests [133]. Several web-based attacks were defined using their state transition definition language and analyzed. Utilizing their WebSTAT state tracking software in conjunction with an Apache web server, a stateful HTTP monitoring with a throughput of almost 100 Mbps was achieved.

The Bro system [99] utilizes a state processing engine to scan for regular expressions. This engine dynamically creates deterministic finite automata (DFA) states whenever a transition occurs into a state which does not already exist. Bro is able to reduce the amount of system resources required for state processing because many states in the DFA will never be reached. The Bro system has been extended to perform stateful processing of bidirectional network flows [117]. Their tests involved the processing of disk files

containing captured network data and tracking the total time required to process all of the captured packets. The modified Bro system was measured to process a 667MB file in 156 CPU seconds, which gives an effective average throughput of 34Mbps.

The effectiveness of the Snort and Bro systems during heavy traffic loads was measured by Lee *et al* [70]. Their experiments showed that both systems become overloaded and fail to detect intrusions when traffic loads exceed 40 Mbps. In order to improve performance, the authors outline adaptive configurations which can be tuned dynamically in response to differing traffic patterns. This type of adaptive IDS increases the probability that an intrusion will be detected during heavy traffic loads.

Intrusion detection systems scan network packets searching for digital signatures corresponding to Internet worms and computer viruses. Some of the more sophisticated intrusion detections systems perform TCP stream reassembly in order to scan for signatures across packet boundaries. To accomplish this task, they need to perform similar flow processing to that of the TCP-Processor. Existing intrusion detections systems are not capable of processing data at multi-gigabit data rates nor are they capable of processing millions of simultaneous TCP connections.

3.5.4 TCP Offload Engines

The main purpose of a TCP Offload Engine (TOE) is to enhance system performance by processing TCP/IP protocol layers in an external device. This allows the main processor to concentrate on application specific processing. As network speeds increase, a disproportionately high processing load will be placed on the main CPU when processing protocols and communicating over these network links [25]. This is attributable to the large number of interrupts handled by the main CPU and the protocol processing overhead incurred per transmitted data byte. When using a normal Maximum Transmission Unit (MTU) size of 1500 bytes, current computer systems which process TCP in software are unable to saturate a Gigabit Ethernet link [96]. A TOE equipped Network Interface Card (NIC) will allow computer systems to communicate efficiently on future high-speed networks.

Due to the negative impact of software protocol processing on high-speed networks, there has been a recent flurry of research activity [138, 50, 102] in the area of TCP offload engines, resulting in two main research directions: placing a computing resource on the NIC which handles protocol processing, and reserving one CPU in a multi-CPU system for performing protocol processing. This activity stems from the realization that TCP processing consumes a large portion of a main processor's resources when dealing with high

traffic rates. When data rates exceed 100Mbps, over half of the CPU processing time can be utilized while processing interrupts, copying data, processing protocols, and executing kernel routines. A TCP offload engine performs protocol processing operations, freeing up the main processor to perform application level processing and user-assigned tasks.

Sabhikhi presents a discussion of the issues surrounding high performance TCP implementations [106]. These issues include managing large numbers of connections, processing data on high speed links, handling high connection rates, limiting instruction memory size, providing fast efficient access to memory, and selecting a programming model (synchronous versus asynchronous, managing data movement, and maintaining the many required timers). Four competing technologies are: (1) general purpose processors, (2) network processors, (3) TCP processors, and (4) network storage processors. General purpose CPUs are over burdened with processing protocols. TCP processing in software typically requires 2GHz of processing power to manage an average mix of packets on a heavily loaded 1Gbps link [10]. This severely reduces the number of compute cycles available for application processing. Memory bandwidth, I/O throughput, and interrupt processing of general purpose CPUs combine to limit overall throughput. Specialized network processors, such as those contained in the Intel IXP-1200 and the Motorola 8260 can perform only limited protocol work. These devices are ideal for handling lower layer networking protocols, but do not have the performance or resources to process TCP. Since these cores execute software instructions, they require extremely high clock frequencies in order to achieve the performance necessary to process traffic at high data rates. Network storage processors support the termination of TCP so that storage devices can be placed anywhere in the network while utilizing existing networking infrastructure. These processors are targeted at providing high availability and high performance storage solutions utilizing existing protocols. The addition of TCP-based communications to network storage processors is a new concept and commercial devices that use it are under development.

Intelligent Network Interface Cards (NICs) will support the processing of higher level protocols in the future. The current focus of development on TOEs is on adding TCP processing engines to NICs that operate at gigabit line speeds or higher. Many of the NIC vendors, including Intel, NEC, Adaptec, Lucent, and others, are actively working on this technology. These devices typically contain a microprocessor and high-speed memory in order to perform TCP state processing and stream reassembly. Data payloads can then be Direct Memory Access (DMA) transferred into main memory for application level processing. Very high performance embedded microprocessors are being developed specifically for the task of offloading TCP processing [138, 50]. These microprocessors operate

at 10GHz clock frequencies in order to processes TCP connections on 10Gbps networks. These processors contain specialized instruction sets which are optimized for the task of performing protocol processing operations. CAMs are utilized in packet classification and packet resequencing operations utilizing on-chip memories. Currently, the TCP offload engine from Intel [138] is unable to process more than 8,000 active flows simultaneously.

The second approach that supports TCP termination on high speed networks is allocating one CPU in a multi-CPU system solely for the task of processing network protocols. Regnier *et al*. have made significant performance gains by partitioning processors in this manner [102]. By segregating protocol processing operations and locking down tasks to specific processors, the operating system avoids contending with issues such as resource contention, lock management, scheduling, and the purging of instruction and data caches. The stack, driver, and kernel accounted for over 96% of the CPU utilization on a 2 processor SMP in a normal configuration. By reserving one processor to handle protocol processing, this percentage was reduced to 53%.

Compared to the TCP-Processor, the main limitation of current generation TCP offload engines is their inability to support large numbers of active flows simultaneously. Additionally, these devices are concerned with terminating a TCP connection and therefore do not provide the enhanced flow processing capabilities of the TCP-Processor. Similar to the TCP-Processor, they perform flow classification and TCP stream reassembly operations, and interact with data processing applications. Unlike the TCP-Processor, TCP offload engines contain additional protocol processing logic required to terminate TCP connections. The TCP flow classification and processing techniques employed by the TCP-Processor are directly applicable to TCP offload engines.

3.6 Hardware-Accelerated Content Scanners

The TCP-Processor is designed to integrate with various types of stream processing applications. Hardware-accelerated content scanners are ideal applications for integration because of their speed of operation and their ability to detect specific content within a TCP flow or apply a regular expression to the TCP data stream. This section describes recent research relating to hardware-accelerated content scanners. The following content scanning architectures are organized in chronological order, with earlier work appearing first and most recent research appearing last.

Sidhu *et al*. developed an FPGA circuit that detects regular expression matches to a given input stream using Nondeterministic Finite Automata (NFAs) [114]. Their algorithm

minimizes time and space requirements in order to improve the efficiency of the implementation. NFAs are implemented as configured logic on an FPGA device. The performance of their hardware-based content scanning architecture is dependant on the length of the regular expression being evaluated. Their system is faster than best case software performance and orders of magnitude better in worst case scenarios.

Franklin *et al*. developed an FPGA-based regular expression compiler in Java using JHDL [41]. They extended the work of Sidhu *et al*. implementing NFAs in reconfigurable hardware and analyzed the regular expressions found in the Snort [104] rule database. For small test cases, the performance of software implementations is similar to that of their hardware implementation. For larger regular expressions, the hardware circuit had a 600x performance improvement over software-based systems.

Cho *et al*. created a rule-based system where rules are transformed into VHDL structures [17]. These VHDL structures are loaded into FPGAs and processed in parallel to exploit the independent nature of different rules. Signature matching is accomplished via four parallel 8-bit comparators. Their implementation is capable of processing data at over 2.88 Gbps. This system has scaling problems because rule matching is performed directly in FPGA logic and this logic area increases at a quadratic rate compared to the number of rules processed.

Moscola *et al*. implemented a system which employs Deterministic Finite Automata (DFA) to scan individual network packets for digital signatures. Their analysis indicates that most regular expressions can be implemented with fewer states using DFA-based implementations than with NFA-based implementations. Operating four scanners in parallel, their circuit is capable of scanning network traffic at 1.184 Gbps. The performance of this system degrades as the number of regular expressions increases. The reduction in performance is caused by signal fan-out delays when routing the input stream to multiple matching engines.

Dharmapurikar *et al*. introduced a hardware circuit which uses a hashing technique to detect fixed-length strings in a data stream [29]. This technique is based on Bloom filters, first described in 1970 [9]. The system uses on-chip static RAM to hold hash tables which support single clock cycle memory accesses. The Bloom filter hashing technique for string searching is subject to false-positives where a match is indicated where none exists. To prevent false positives, additional logic is required to validate every match. The system is capable of scanning for 10,000 strings at 2.4 Gbps.

Clark *et al*. developed an FPGA circuit capable of analyzing complex regular expressions against an arbitrary input data stream [21]. The design seeks to maximize pattern

matching capacity and system throughput by reducing the circuit area required to perform the character matches. The system is capable of scanning for 17,000 characters in over 1,500 rules of the Snort rule database at gigabit traffic rates.

Sourdis *et al*. employ fine grain pipelining in FPGA devices to perform high-speed string matching [118]. They utilize multiple comparators and parallel matching to maintain high throughput. A packet fan-out tree is implemented, allowing every rule to receive the incoming packet at the same time. This improves the overall throughput of their system. By performing four comparisons on each clock cycle, each shifted by one byte, their design can scan 32 bits per clock cycle. A maximum throughput of 11 Gbps was achieved when scanning for 50 Snort rule on a Virtex2 1000 FPGA device.

Baker *et al*. developed a linear-array string matching architecture for use in reconfigurable hardware [4]. Their method uses a buffered, two-comparator variation of the KMP [62] algorithm. The system is capable of processing data streams at 1.8 Gbps. Small memory banks hold string patterns and jump tables which can be reconfigured at run-time, without affecting the throughput of the system.

Lockwood *et al*. built a reprogrammable firewall using DFAs and CAMs implemented in FPGA logic [77]. This system scans packet payloads using regular expressions and processes a small number of packet headers using Ternary Content Addressable Memories (TCAM). The reprogrammable firewall, without the content scanning component, can process data at 2 Gbps. The regular expression processing engine only operates at 296 Mbps. This performance can be increased by instantiating multiple copies of the regular expression processing engine.

Sugawara *et al*. developed a multi-stream packet scanner [122]. The algorithm is based on the Aho-Corasick algorithm which uses a trie structure that corresponds to the longest pattern match [61]. A small amount of context information is maintained which pertains to the current matching state of a flow. This context information can be swapped in and out of the scanner. An implementation of this algorithm is capable of processing data at 14 Gpbs when scanning for 2000 characters.

While the TCP-Processor is designed to integrate easily with any type of TCP stream processing application, hardware accelerated content scanners are ideal integration candidates because of their ability to process large quantities of data at very high speeds. The majority of these research efforts focus on context scanning algorithms which operate on a single data flow. The work by Sugawara *et al*. [122] focused on developing a content scanner which minimizes per-flow context information. This simplifies the process of storing and retrieving scanning state so that multiple data flows can be processed by a single

scanning unit. A content scanner capable of quickly swapping scanning state is most likely to realize the benefits of high-speed flow processing by integrating with the TCP-Processor.

3.7 Summary

The TCP-Processor describes a new type of multi-context TCP flow processor that is capable of operating on the high-speed network links of today and tomorrow. The TCP-Processor architecture overcomes the limitations of software-based implementations by using multiple levels of pipelining for packet processing operations and exploiting fine grain parallelism. The architecture is implemented in FPGA logic and does not incur any operating system overhead. To improve the throughput and performance of the system, packets are not stored in external memory during normal processing.

The TCP-Processor also provides distinct advantages over other hardware-based TCP processors. The TCP-Processor's main distinction from other research is its ability to handle millions of simultaneous TCP flows. This is accomplished by maintaining a small amount of per-flow context information stored in external memory which can quickly be swapped in and out of the TCP state processing engine. Another of the TCP-Processor's advantages is its extensibility. Its modular architecture can be extended or modified to incorporate different algorithms and new features. The application interface annotates network packets which provides stream processing applications with full packet headers in addition to TCP stream data.

Chapter 4

Architecture

This chapter provides an overview of the TCP processing architecture. The first section reviews the initial phases of research in order to provide a historical context for the development of the TCP-Processor. This summary includes brief descriptions of the TCP-Splitter circuit and early work on the TCP-Processor, which was incorporated into the StreamCapture circuit. The next section provides an overview of the TCP-Processor which describes its underlying components, their basic operation, and the data flow through the circuit. The following section describes the application interface for the TCP-Processor, providing definitions for each of the interface signals and an explanation of the interface waveform. Finally, the chapter provides information on the extensibility of the architecture along with a description of how this architecture can be employed in complex flow processing applications which require coordination among multiple FPGA devices.

4.1 Initial Investigations

This research work responds to the challenge of processing the data for large numbers of TCP connections within the context of a high-speed network. This challenge is greatest when dealing with the real-time packet processing constraints related to the computation and memory operations required to analyze a single TCP packet. Existing TCP processing systems, such as those that detect and filter Internet worms and computer viruses, have severe performance limitations which prevent them from operating in this environment. To be effective, data processing systems must have access to an accurate representation of the application data transiting a network. Reconstructing byte streams from individual packets is required when processing data contained in TCP packets.

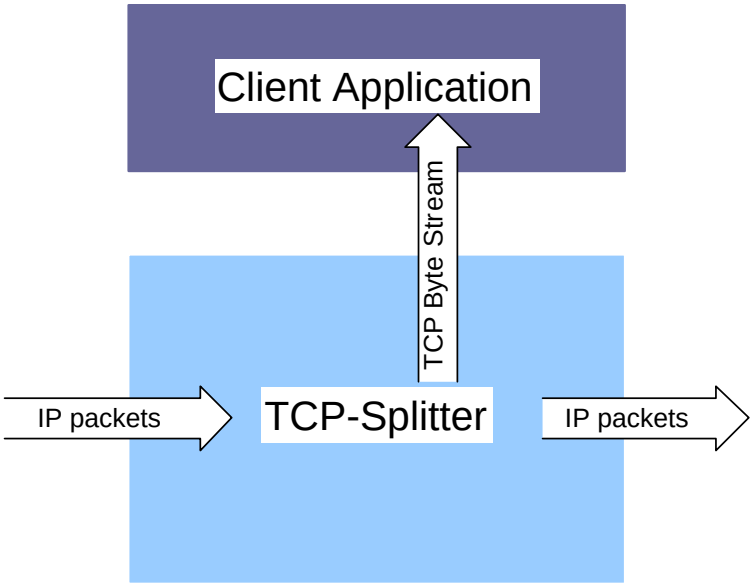


Figure 4.1: TCP-Splitter Data Flow

Field Programmable Gate Arrays (FPGAs) contain large numbers of reconfigurable logic units interconnected with an underlying routing fabric. These devices support parallel operations, pipelined operations, and provide tightly-coupled access to high-speed memories. The initial research associated with this dissertation originated with the idea of using FPGA devices to process TCP data flows in high-speed networked environments.

4.1.1 TCP-Splitter

Initial investigations of this problem resulted in a proof of concept design called the TCP-Splitter [110, 109]. This technology splits the network data stream into two separate copies, forwarding one on to the next hop router or end system and another copy to the monitoring application. Figure 4.1 shows the flow of IP packets through the TCP-Splitter circuit and the delivery of a TCP byte stream to a client application.

A produced and tested hardware circuit based on the TCP-Splitter design supports the processing of 256 thousand TCP flows at 3.2Gbps. The TCP-Splitter maintains 33 bits of context information per flow. This context data includes a one bit indication of whether or not the flow is active and a 32 bit number specifying the next expected TCP sequence number for connection.

A multiple device programmer [108, 65] showcases the viability of the TCP-Splitter technology. This circuit extracts device configuration and programming information from

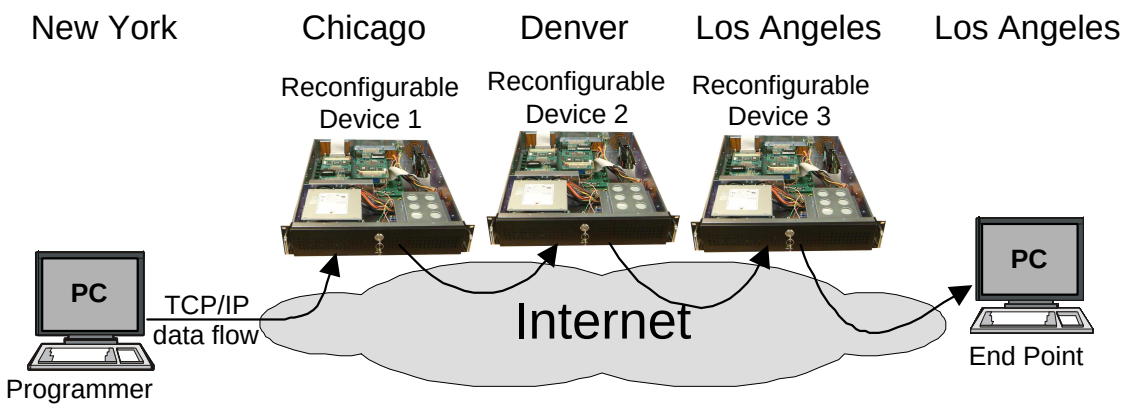


Figure 4.2: Multi-Device Programmer Using TCP-Splitter Technology

a TCP connection and uses this information to program attached devices. Since the TCP-Splitter processes network packets, remote devices can be programmed using this technology as long as there is network connectivity between the device and the programming station. Figure 4.2 shows an example of a programmer located in New York programming devices in Chicago, Denver, and Los Angeles. In this example, a TCP connection is established with a computer acting as the connection end point. The TCP packets sent from the programming computer to the end point computer travel through the intermediate locations containing the devices to be reprogrammed.

The TCP-Splitter design contains several limitations which are addressed by the TCP-Processor. These limitations are listed below:

- The number of TCP flows which can be concurrently processed is limited to 256 thousand. High-speed network communication links that operate at 2.5Gbps data rates and above can support millions of simultaneous TCP connections. In order to operate effectively in this environment, the TCP-Splitter needs to be able to handle a larger number of concurrent connections. Because the TCP-Splitter uses limited capacity SRAM devices for per-flow context storage, the TCP-Splitter has a bounded limit on the number of simultaneous connections it can process relative to maximum size of the available SRAM memory devices.

- The TCP-Splitter technology has a small amount of memory for storing per-flow context information. It uses a 1MByte memory module to store per-flow context information. Each memory location is 36-bits wide and represents the context storage associated with a single TCP flow. This amount of memory is insufficient to store

proper per-flow context information. Realistic storage requirements are in the tens of bytes range.

- Hash collisions are not handled within the per-flow context storage area. The memory device is too small to support direct addressing using a 96-bit index obtained by concatenating the source IP address, the destination IP address, the source TCP port and the destination TCP port. In addition, there is insufficient memory available to store these items within each context record. Because of these memory limitations, the TCP-Splitter cannot tell the difference between two TCP flows that hash to the same memory location. When this situation occurs, the TCP-Splitter will continue to process the first flow and treat packets from the second flow as either retransmitted packets or as out-of-sequence packets.

- The TCP-Splitter only allows client applications to operate as passive monitoring devices. This processing model is inadequate for classes of applications which need to manipulate network traffic. Intrusion prevention systems are one such example. Without the ability to alter, drop or generate network packets, a system of this nature would not be able to stop the spread of malicious content.

- There is no support for the reordering of out-of-sequence packets. The TCP-Splitter actively drops selective out-of-sequence packets from the network, counting on the end hosts to detect and retransmit the missing data. This act of dropping out-of-sequence network packets forces data to flow through the network in the proper sequence. Data is normally transmitted through the network in order. Under ideal conditions, the packet dropping behavior of the TCP-Splitter has no effect on network traffic. When route changes, network congestion, fiber cuts, router overloads and other disruptions occur to the normal flow of network traffic, this packet dropping behavior can have a severe negative impact of the performance of the network. For this reason, the TCP processing circuit needs to support the re-ordering of out-of-sequence packets without removing packets from the network.

4.1.2 StreamCapture

The StreamCapture circuit attempts to address the limitations of the TCP-Splitter technology while adding enhanced processing features. This circuit is designed to be used in a single chip implementation where the TCP processing logic and the client application reside on the same FPGA device. The close proximity between these circuits allows the

StreamCapture circuit to provide context storage services to the client application without requiring additional memory accesses.

The StreamCapture circuit also supports blocking and unblocking TCP data streams at a client application-specified TCP sequence number. This feature is useful to applications which need to block the flow of data on a particular connection while authorization and validation operations take place. Once access has been granted, the flow of data is re-enabled for that connection. There is also support for the termination of an active flow. This can be accomplished by generating the appropriate TCP reset (RST) packets and transmitting them to both end systems.

The StreamCapture circuit maintains 64 bytes of context for each TCP flow. Context required for TCP processing is stored in the first 32 bytes of this area and includes the IP and TCP header information required to uniquely identify this flow. This information is used to differentiate separate TCP connections from each other when hash collisions occur in the state store. To ease the processing burden on client applications, the StreamCapture circuit also maintains application-specific context information in the other 32 bytes of the per-flow record. Using burst read operations, this extra information can be retrieved from external memory with a minimal amount of overhead.

In practice, maintaining per-flow context information for client applications was more of a liability than a feature due to the time delay between the storage and retrieval of application-specific context information. Figure 4.3 shows the layout of the StreamCapture circuit. All per-flow context information is stored in an external memory device. When processing a packet, the TCP Processing Engine retrieves application-specific context information and passes it to the client application with the network packet. After processing the packet, the client application passes updated context information to the Enhanced Flow Services module, which then updates external memory. To achieve high throughput, the StreamCapture circuit is pipelined so different modules simultaneously process different network packets. The time delay between the retrieval of application-specific context information from external memory and the updating of this context information leads to inconsistencies in data processing. To resolve this issue, the StreamCapture circuit can suspend processing until the application-specific context information is updated. This reduces the throughput of the circuit and can lead to dropped packets when buffers overflow. Another possible solution requires the client application to maintain a cache of recently processed flows. Since the client application has no direct access to external memory, this configuration can lead to cache coherency problems and processing packets with improper context information.

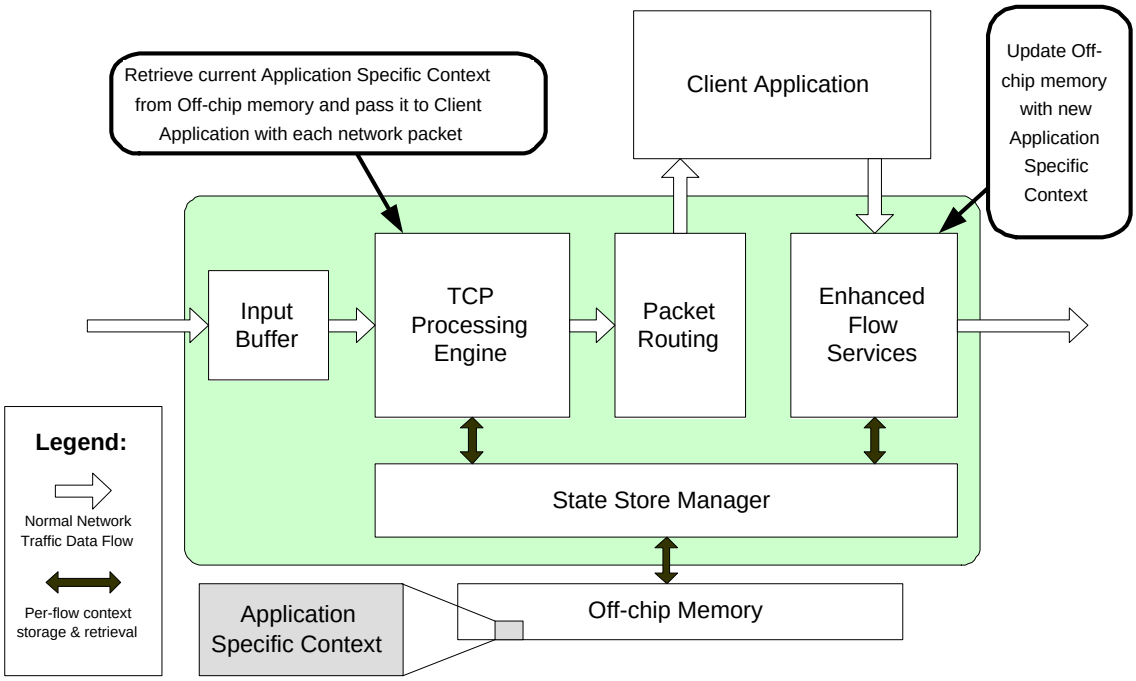


Figure 4.3: StreamCapture Circuit

4.2 TCP-Processor

The architecture of the TCP-Processor contains a feature set derived from the preceding work on TCP processing circuits. The feature set includes:

- Maintenance of 64 bytes of per-flow context for each TCP connection.
- Support for the processing of millions of simultaneous TCP connections.
- Easy integration of other classification and lookup algorithms.
- Processing of network traffic at OC-48 (2.5Gbps).
- A scalable design which can grow to support higher speed networks of the future.
- Allowance for the processing of bi-directional TCP flows by associating packets in opposite directions of the same flow with each other.

The TCP-Processor consists of seven separate components: an input buffer, a TCP processing engine, a packet storage manager, a state store manager, a packet router, an egress processing component and a statistics component. This is shown in Figure 4.4. The interfaces between each of these components are designed to simplify module integration, modification and replacement tasks. In addition, integration efforts can utilize these internal subcomponent interfaces to extend the capabilities of the TCP-Processor.

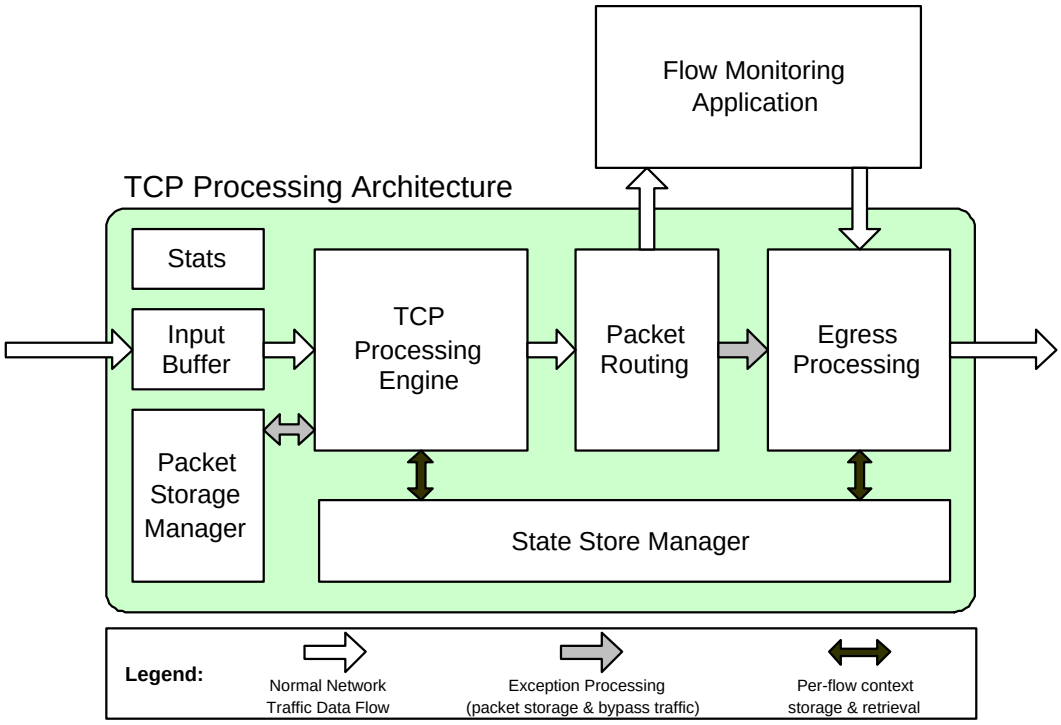


Figure 4.4: TCP-Processor Architecture

The input buffer processes data as it enters the TCP-Processor, providing packet buffering services during periods of downstream delay. Memory contention, management overhead and flow control resulting from downstream congestion can cause these processing delays. In addition, the client application has the ability to induce back pressure into the TCP-Processor by driving its own flow control signal and cause additional delays. The input buffer component also ensures that any data loss in the circuit occurs in whole packet increments, thus protecting the rest of the circuit from having to process incomplete packet fragments.

Data from the input buffer flows into the TCP processing engine, which manages protocol processing operations. This module computes TCP checksums, tracks flow processing state, and performs computations related to the flow classification operation. In addition, it communicates with a packet store manager for storing packets. This provides a slow path through the circuit where packets can be stored and later reinjected for continued processing.

The state store manager manages all of the per-flow context storage and retrieval operations through communications with the memory controller. It exposes a simple interface to the flow classifier and the egress processing components for retrieving and storing

context information. This modular design allows for the replacement of the memory subsystem to support other memory devices or different lookup algorithms. The state store manager communicates with large memories which hold the per-flow context information for the TCP processing engine.

The packet storage manager provides packet buffering services for the TCP processing engine. Packets received in an improper order by the TCP processing engine can be passed off to the packet storage manager until packets containing the missing data arrive. At that point, the stored packets can be re-injected into the TCP processing engine. Consider the packet sequence 1, 2, 4, 5, 3. The packet storage manager stores packets 4 and 5 until packet 3 is received. At that time, packets 4 and 5 are re-injected into the data path. This processing ensures that data is passed to the monitoring application in the proper sequence.

The packet routing module determines where the current packet should be directed. There are three possible routing choices. The normal flow of traffic is routed to the monitoring application component, where it is processed. Retransmitted frames and other non-TCP-based packets can either be routed with the monitored TCP traffic to the monitoring application or forwarded directly on to the egress processing module for transmission to the end system or next hop router. Packets with invalid checksums are dropped from the network. When the TCP-Processor is configured for in-order processing, packets which reach this point whose sequence numbers are greater than what is expected by the TCP processing engine are dropped in order to ensure that TCP flows traverse the network in order.

The egress processing module receives packets from the monitoring application and performs additional per-flow processing. If so directed by the monitoring application, the egress processing module can block a selected flow by ensuring that packets which contain data exceeding a specified blocking sequence number will be dropped from the network. In this manner, a network flow can be temporarily blocked and re-enabled. The termination of a selected flow can also be performed at this time by blocking the flow and generating a TCP RST packet. TCP checksums are validated and can be corrected in situations where the client application has modified packets or inserted new packets into the data stream. Bypass packets received directly from the routing module are interleaved with packets received from the monitoring application for transmission to the outbound interface. Statistics packets are also interleaved into the outbound traffic.

All of the TCP-Processor's components supply event triggers to the statistics component. Internal counters accumulate the occurrence of these event triggers, providing a

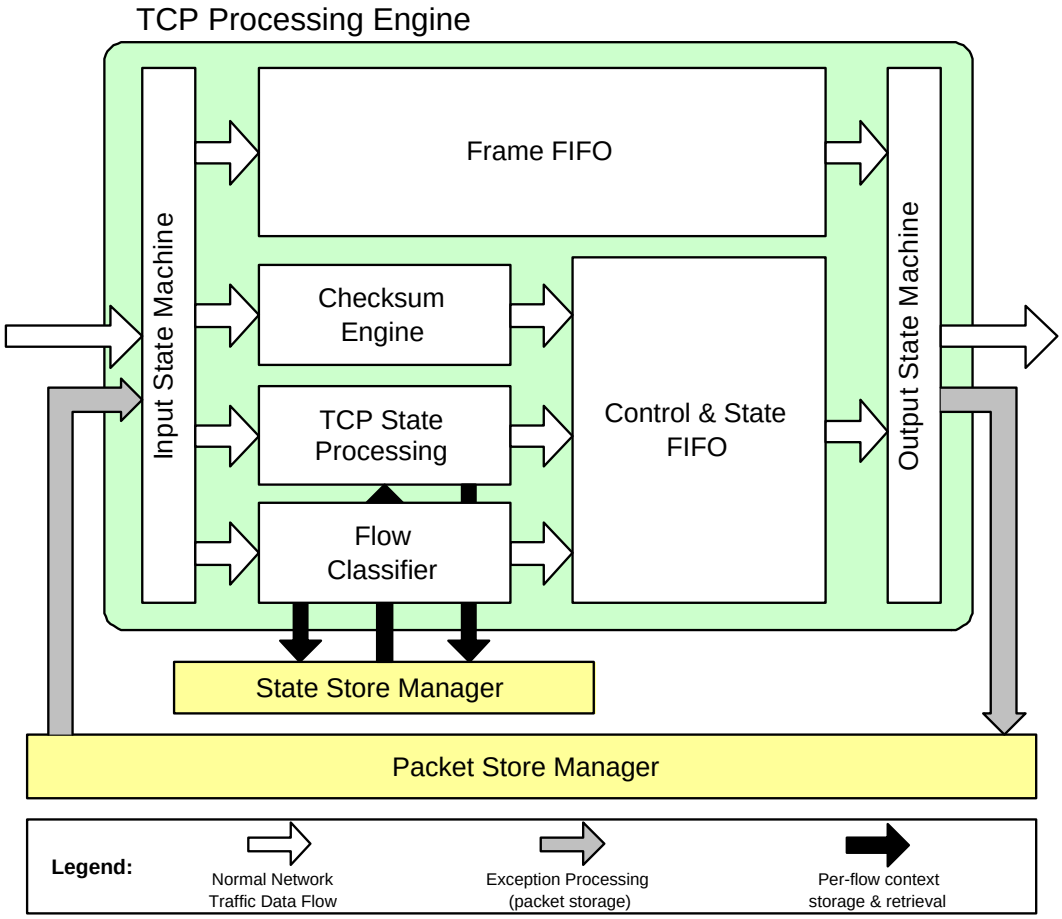


Figure 4.5: TCP Processing Engine

collection point for statistical information. On a periodic interval, the these counter values are collected and inserted into a User Datagram Protocol (UDP) packet and passed to the egress processing component. At this same time, the individual counters are reset to zero and begin counting events in the next collection period.

The TCP processing engine consists of seven subcomponents. Figure 4.5 illustrates a detailed view of the organization of these subcomponents. The input state machine is the first to process data, provides the operating state, and keeps track of the current position within each packet. It also passes state information and packet data to the frame FIFO, checksum engine, TCP state processing and flow classifier components. The frame FIFO buffers packets while other computations are taking place. The checksum engine computes the TCP checksum to validate the correctness of the TCP packet. The TCP state processing module keeps track of the TCP processing state and determines whether packets

are in-sequence or out-of-sequence. The flow classifier communicates with the state store manager to save and retrieve per-flow context information. Results from the TCP state processing module, the checksum engine, and the flow classifier are written to a control FIFO. The output state machine extracts data from the control and frame FIFOs and passes it to the routing module. The results of the protocol processing operations are delivered to downstream components synchronously with the start of each packet. The internal FIFOs allow the inbound processing modules to work on the next packet while the outbound module processes earlier packets. The interfaces to the flow classification subsystem are designed for speed and efficiency while still supporting interchangeable classifiers. In this manner, future enhancements to flow classification algorithms can easily be incorporated into this architecture by only replacing the required component. This flexibility also supports the testing of various classification algorithms.

4.3 Application Interface

One of the key goals of the TCP-Processor architecture is to provide a simple, but complete mechanism for accessing TCP stream content that can be used by a wide range of extensible networking applications. A clean interface hides the intricacies of the underlying protocol processing and reduces the complexity associated with processing the TCP data streams. The signals listed below are used to transfer TCP stream data to the monitoring application:

DATA 32 bit vector (4 byte wide) main data bus which carries all packet data and lower layer protocol information.

DATAEN 1 bit signal indicating whether or not there is valid data on the DATA lines.

SOF 1 bit signal indicating the start of a new frame (packet). This signal is always asserted in unison with the first word on the DATA bus.

SOIP 1 bit signal indicating the start of the IP header within the packet.

SOP 1 bit signal indicating the start of the IP payload (ie. the first word past the end of the IP header). This signal can be used to indicate the first word of the TCP header.

EOF 1 bit signal indicating the end of the frame (packet).

TDE 1 bit signal indicating whether or not there is TCP stream content on the DATA lines.

BYTES 4 bit vector indicating which of the four bytes on the DATA lines contain valid TCP stream data that should be processed by the monitoring application. One bit is used for byte of the DATA word.

FLOWSTATE 32 bit vector (4 byte wide) per-flow state information bus which carries the flow identifier, next TCP sequence number, TCP data length, and a flow instance number.

FLOWSTATEEN 1 bit signal indicating whether or not there is valid data on the FLOWSTATE lines.

NEWFLOW 1 bit signal indicating that this packet represents the start of a new flow. This signal is asserted for one clock cycle in unison with the SOF signal.

ENDFLOW 1 bit signal indicating that this packet represents the end of an existing flow. This signal is asserted for one clock cycle in unison with the SOF signal.

TCA 1 bit flow control signal which the client application can use to suspend the flow of data. It is always assumed that the TCP-Processor can finish sending the current packet.

The client interface contains network data packets and control signals which annotate the packets to the monitoring application. This simplifies the logic required to process TCP data streams while providing the application with access to the complete network data packet. Monitoring applications will likely require access to protocol header fields in order to determine things like the source and destination addresses. Passing the full contents of the packet eliminates the need for extra interface signals which provide this type of data. Control signals assist the monitoring application in navigating to specific locations within the packet. For instance, if the monitoring application were interested in obtaining the source and destination TCP ports, instead of tracking the protocol headers, the SOP signal indicates that the TCP ports are on the DATA lines.

The TCP-Processor maintains per-flow context information for the TCP flow monitoring service. The FLOWSTATE bus contains this additional flow context information which is passed to the monitoring application at the start of each packet. This predefined sequence of data includes a unique flow identifier, the next expected TCP sequence number following the processing of this packet, the TCP data length in bytes, and a flow instance number which can be employed to prevent conflicts when flow identifiers are reused.

Figure 4.6 includes a timing diagram showing the transmission of a single TCP data packet to the monitoring application. The DATA and FLOWSTATE busses contain an indication of the data that is present at each clock cycle during normal processing. The various components of the IP and TCP headers are shown with the control signals. In this example, the NEWFLOW signal indicates that the data contained within this packet represents the first few bytes of a new data stream and that the monitoring application should initialize its processing state prior to processing this data. Additional information is

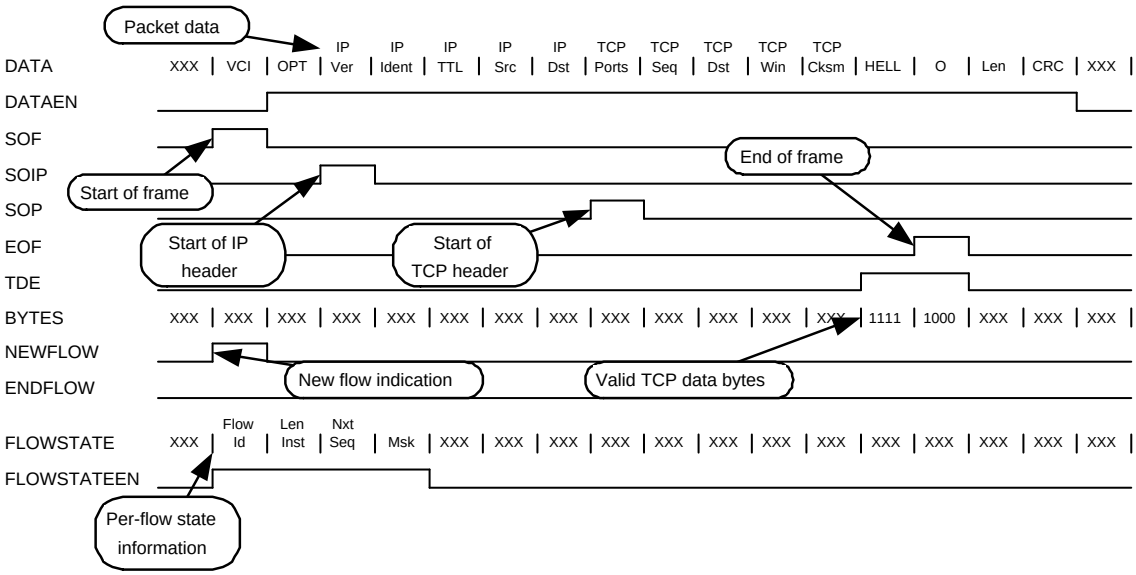


Figure 4.6: Timing Diagram showing Client Interface

clocked through the data bus before and/or after the packet data. This allows for lower-level protocols, such as Virtual Circuit Identifier (VCI), Ethernet Media Access Control (MAC) addresses, other lower layer protocol information, or ancillary data (like packet routing information) to pass through the hardware along with the packet. For example, the AAA data value represents one word of control information prior to the the start of the IP packet. The BBB and CCC labels represent trailer fields that follow the IP packet. The AAA, BBB, and CCC content should be ignored by the monitoring application and passed through to the client input interface of the TCP-Processor for use in outbound packet processing.

4.4 Extensibility

The TCP-Processor contains a modular set of components which support integration with other technologies. This enables development of new components that simply plug into the existing inter-component interfaces.

As chip-making technology advances, the TCP-Processor architecture can scale to operate within devices that exploit higher gate densities and faster clock frequencies. It was designed using a 32-bit wide main data path for network traffic which will easily port to future generations of FPGA and ASIC devices maintaining this property. Increasing the width of the main data path to 64 or 128 bits could double or quadruple the throughput

of the circuit, although it would require reworking the protocol processing subsystems, widening of internal buffering FIFOs, and modifying interface specifications.

The state store manager supports the easy implementation of alternative flow classification algorithms. This allows a design engineer using the TCP-Processor to specify the algorithm which most closely matches the requirements of their TCP flow monitoring service. In addition, the state store manager can be modified to utilize different types of memory managers and cache subsystems.

Interaction with external memory is separate from the logic of the state store manager. This enables an easy migration path to future performance enhancing improvements in technology. As the name suggests, Double Data Rate (DDR) memories support two memory operations per clock cycle thereby doubling the rate at which data can be transferred to/from external memory devices. Increasing the data bus width when communicating to external memories will improve memory performance, but will require additional changes to the state store manager to support the larger bus size.

4.5 Multiple FPGA Coordination

Complex monitoring applications may require multiple hardware devices to perform additional TCP flow processing functions. For example, the whole Snort [104] rule database contains approximately 2000 rules, each with header and content match criteria. Due to the size of this rule set, complete processing of the full rule set cannot be performed within a single FPGA device. Fortunately, Snort rule processing can be broken down into three distinct logical steps: (1) header processing, (2) content scanning, and (3) rule matching. The complete process can be implemented by assigning these separate tasks to multiple hardware devices.

Figure 4.7 illustrates two different configurations of a hardware-based Snort implementation which utilize multiple hardware devices. The first demonstrates a routing configuration where higher levels of flow processing occur above the TCP-Processor. The monitoring application passes data back to the TCP-Processor for final processing which forwards it on to the end station or next hop router. In this configuration, each device provides a reverse channel so data can be passed back the the TCP-Processor. This is the normal data path for the TCP-Processor.

The second example shows a straight-through routing configuration where the TCP processing engine consists of two components: one for ingress processing and one for egress processing. These two components exist on separate devices to enable data to be

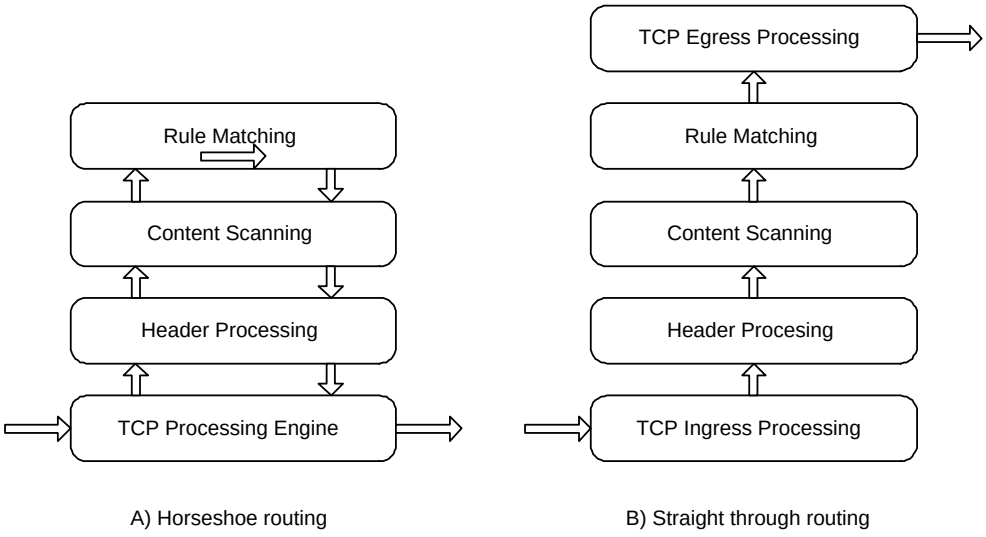


Figure 4.7: Multi-Board Traffic Routing

routed linearly through multiple devices. The advantage of this type of configuration is that data is always moving in a single direction.

A set of flexible serialization and deserialization components efficiently moves data between devices when data processing functions are partitioned across multiple devices. All of the data presented at the TCP-Processor client interface is serialized into an encoded data format which is transmitted to other devices. The deserialization process, in turn, converts this encoded data back into the client interface signals of the TCP-Processor. Figure 4.8 shows a layout of the TCP-Processor circuit components which support multiple board TCP flow monitoring. The dotted line indicates that the ingress portion of the TCP-Processor circuit can be separated from the egress portion of the circuit; a useful feature in straight-through processing configurations.

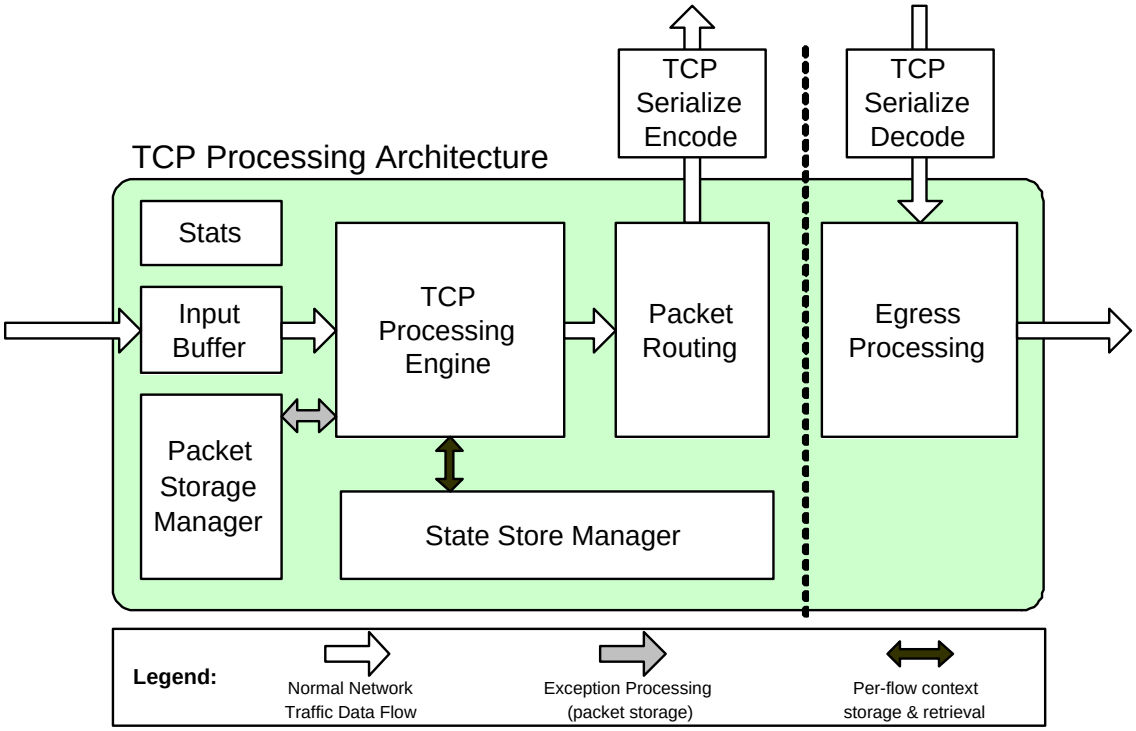


Figure 4.8: Circuit Configuration for Multi-Board Operation

Chapter 5

Environment

This dissertation presents research performed at the Applied Research Laboratory (ARL) at Washington University in St. Louis. Development of the TCP-Processor relied on previous research performed at this facility. The following sections briefly describe some of the hardware, software, and circuit designs which contributed to the development of the TCP-Processor.

The TCP-Processor was designed, implemented and tested within the context of the Field-Programmable Port Extender (FPX). While the circuit is not limited to this specific hardware environment, minor modifications to the TCP-Processor will most likely be required for operation on a different hardware platform. All of the research relating to this dissertation used the FPX platform as the underlying hardware environment.

5.1 Field-programmable Port Extender

The Field-programmable Port Extender (FPX) [74, 73] is a research platform that supports the processing of high-speed network traffic with reconfigurable devices (Figure 5.1). All processing on the FPX is implemented in reconfigurable logic with FPGAs. One FPGA on the system is called the Reprogrammable Application Device (RAD) and is implemented with a Xilinx Virtex XCV2000E (earlier models of the FPX use the XCV1000E) [78]. The RAD can be dynamically reconfigured to perform user-defined functions on packets traversing through the FPX [124]. The second FPGA device on the FPX is called the Network Interface Device (NID) and is implemented with a Xilinx Virtex XCV600E FPGA. The NID, which is statically programmed at power-up, controls data routing on the FPX

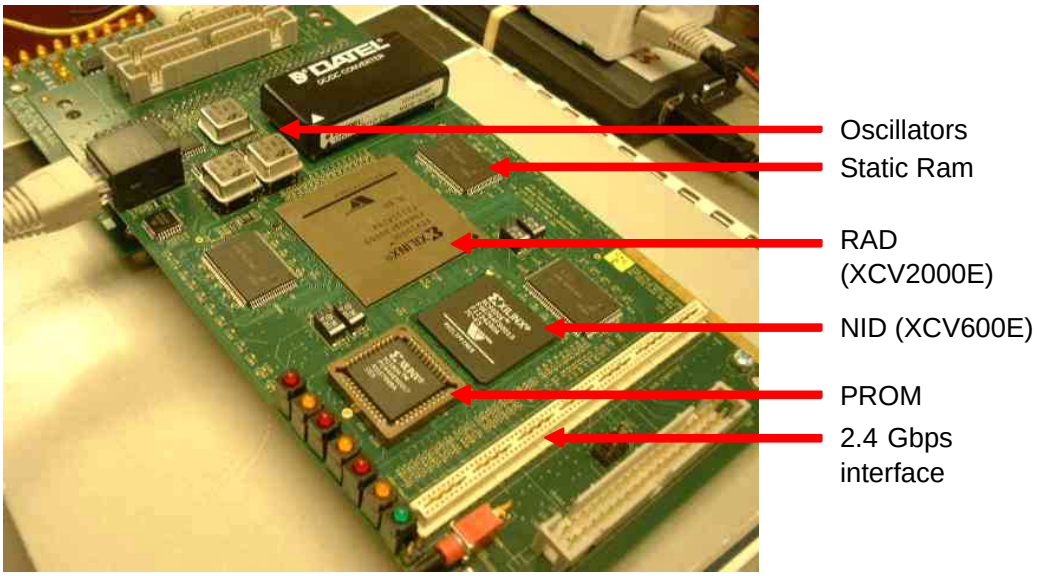


Figure 5.1: Field Programmable Port Extender

platform and is also responsible for programming the RAD. The system contains five parallel banks of memory that include both Zero Bus Turnaround (ZBT) pipelined Static Random Access Memory (SRAM) for low-latency access to off-chip memory and Synchronous Dynamic RAM (SDRAM) to enable buffering of a gigabyte of data. Network interfaces allow the FPX to interface at speeds of OC-48 (2.4 gigabits per second). The RAD device can be remotely reprogrammed by sending specially configured control cells to the device.

The external interfaces of the FPX cards are positioned on the bottom left and top right of each card. This design feature allows the devices to be stacked on top of each other by rotating each successive device 180 degrees, which aligns the external interfaces with each other. Complex network traffic processing applications, like those that scan and remove Internet viruses, can be implemented by coordinating activities amongst multiple stacked FPX devices.

5.2 Washington University Gigabit Switch

The Washington University Gigabit Switch (WUGS) [129] is an eight port ATM switch interconnected by an underlying switching fabric. Data paths on the switch are 32 bits wide. The backplane of the switch drives each of the eight ATM ports at a data rate of 2.4 Gbps to achieve an aggregate bandwidth of 20 Gbps.

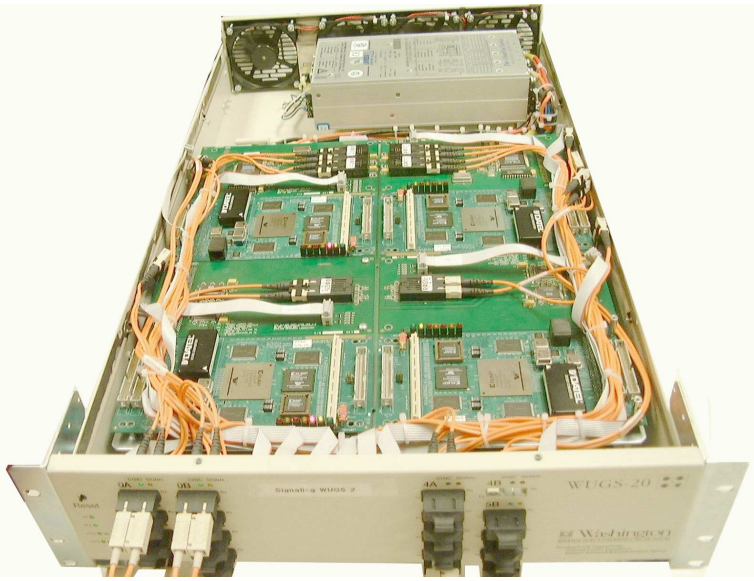


Figure 5.2: Washington University Gigabit Switch Loaded with Four FPX Cards

Each of the eight ports supports several different types of adapter cards. The current set includes a Gigabit Ethernet line card, a Gigabit ATM line card, an OC-3 ATM line card, an OC-48 ATM line card, the Smart Port Card (SPC), the Smart Port Card II (SPC2), and the Field-programmable Port Extender (FPX). The SPC and SPC2 contain a Pentium processor and Pentium III processor respectively, and can execute a version of the Unix operating system which supports software-based packet processing [27]. Figure 5.2 shows a WUGS switch populated with four FPX cards, two OC-3 line cards, and two GLink line cards.

5.3 NCHARGE

A suite of tools called NCHARGE (Networked Configurable Hardware Administrator for Reconfiguration and Governing via End-systems) remotely manages control and configuration of the FPX platform [128]. NCHARGE provides a standard Application Programming Interface (API) between software and reprogrammable hardware modules. Using this API, multiple software processes can communicate to one or more FPX cards using standard TCP/IP sockets.

The NCHARGE tool set also includes a web interface which allows remote access and control of FPX hardware to be accomplished using a standard web browser. The web

Figure 5.3: FPX-in-a-Box System

interface uses a Common Gateway Interface (CGI) script to execute small stub routines which communicate with NCHARGE over the standard sockets interface. This allows the commands issued via the web interface to operate in conjunction with other programs communicating with NCHARGE.

5.4 FPX-in-a-Box

The FPX platform can be used either in conjunction with a WUGS switch or independently as part of a standalone solution. The FPX cards have a network interface port on both the top and bottom, enabling stacking of other FPX or line cards under and over them. The FPX-in-a-box system (Figure 5.3) provides a simple backplane which supports two stacks of FPX cards with a line card placed at the top of each stack. The provides a small footprint system which is capable of performing complex network processing at OC-48 data rates without requiring a underlying switch fabric. Systems needing complex flow processing can be implemented by stacking multiple FPX cards and distributing circuits across the multiple FPGA devices.

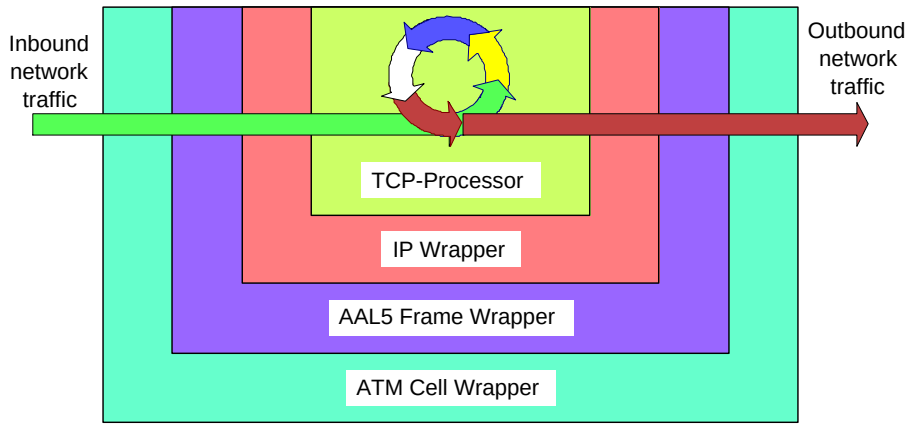


Figure 5.4: Layered Protocol Wrappers

5.5 Protocol Wrappers

FPX cards process Internet packets using a set of Layered Protocol Wrappers [14, 12, 13]. Each protocol wrapper performs processing on a different protocol layer for inbound and outbound traffic. These protocol wrappers create a framework that allows upper layer protocol processing to be isolated from any underlying protocols. The protocol wrapper library includes an ATM Cell wrapper, an AAL5 Frame wrapper, an IP protocol wrapper and a UDP protocol wrapper. The Cell Wrapper validates the Header Error Checksum (HEC) field in the ATM cell header and performs routing based on the cell's Virtual Channel Identifier (VCI) and Virtual Path Identifier (VPI). The Frame Wrapper performs Segmentation and Reassembly (SAR) operations converting cells into packets and packets back into cells. The IP Wrapper performs processing of IP packets. The TCP-Processor circuit interfaces with the IP, Frame and Cell Wrappers for processing network traffic. Figure 5.4 diagrams the protocol processing hierarchy.

The IPWrapper sends and receives IP packets utilizing a 32-bit wide data bus for carrying data and several associated control signals. These control signals include a start of frame (SOF) signal, a start of IP header (SOIP) signal, a start of payload (SOP) signal, and an end of frame (EOF) signal. Packets themselves can be broken down into five sections as described in Table 5.1.

The IPWrapper is hard-coded to process only the traffic that enters on VCI 50 (0x32) or VCI 51 (0x33). Because of this feature, all network traffic processed by the IPWrapper must arrive on VCI 50 (0x32) or VCI 51 (0x33). This includes TCP traffic to be processed by the TCP-Processor and control traffic used to remote command and control various

Table 5.1: IP packet contents

Data	Control signals	Comment
ATM VCI	SOF	first word of packet
preamble	dataen	zero or more words inserted before the start of the packet
IP header	dataen, SOIP, EOF	IP header information
IP payload	dataen, SOP, EOF	IP payload information
trailer	dataen	contains AAL5 trailer fields

circuit designs. In order to process data on a VCI other than 50 (0x32) or 51 (0x33), VHDL code changes will be required in the lower layered protocol wrappers.

Chapter 6

TCP-Processor Internals

The TCP-Processor is a subcomponent of the StreamExtract application discussed in the next section. The TCP-Processor interfaces with the previously described layered protocol wrappers and provides protocol processing for TCP packets. The TCP-Processor is specifically concerned with reassembling application data streams from individual TCP packets observed on the interior of the network. This section will focus on the design, layout and operation of the TCP-Processor.

The source code for TCP-Processor is located in the directory `vhdl/wrappers/TCPProcessor/vhdl` found in the `streamextract_source_v2.zip` distribution file. The following VHDL source modules can be found there and comprise the core of the TCP-Processor technology:

`tcpprocessor.vhd`- Top level TCP-Processor circuit. The external interface includes configuration parameters, clock and control signals, inbound and outbound 32-bit UTOPIA interfaces which provides access to raw ATM cells, two separate SRAM interfaces, a single SDRAM interface, and four output signals tied to LEDs. The TCPProcessor component interfaces the lower layer protocol wrappers with the core TCP-Processor.

`tcpproc.vhd`- This module encapsulates the individual components which make up the TCP-Processor. Network traffic travels between the external interfaces and the various subcomponents.

`tcpinbuf.vhd`- The TCPInbuf component provides inbound packet buffering services for the TCP-Processor when back pressure or flow control is driven by downstream components. This module also ensures that data loss occurs on packet boundaries.

`tcpengine.vhd`- The TCPEngine component performs TCP processing. It communicates with the state store manager to retrieve and store per-flow context information.

`statestoremgr.vhd`- The StateStoreMgr performs context storage and retrieval services. It exposes a simple external interface and handles all complex interactions with SDRAM.

`tcprouting.vhd`- The TCPRouting module routes packets previously processed by the TCPEngine to either the client monitoring circuit, the outbound TCPEgress module, or to the bit bucket. Additional control signals indicate how each packet should be routed.

`tcpstats.vhd`- The TCPStats component collects statistical information from other components and maintains internal event counters. On a periodic basis, it generates a UDP statistics packet and sends it to the configured destination address.

`tcpegress.vhd`- The TCPEgress module merges statistics traffic from the TCPStats component, bypass traffic from the TCPRouting component, and return traffic from the client monitoring application. TCP checksum values can also be regenerated for TCP packets returning from the client monitoring application to support flow modification.

The components of the TCP-Processor directly correspond to the various source files. To simplify the layout of the circuit, each component is completely contained in a single, unique source file. Figure 6.1 diagrams the components of the TCP-Processor and their relative significance in the VHDL hierarchy.

6.1 Endianness

The TCP-Processor was developed with little endian constructs. When accessing memory, the least significant byte is stored at the lowest address. In addition, the least significant bit is the lowest bit of the vector (usually bit zero) in all signal vectors. This is the most intuitive method for representing data and is used throughout the TCP-Processor and associated circuits. The TCP-Processor works with network headers using little endian constructs, even though network byte order is a big endian format.

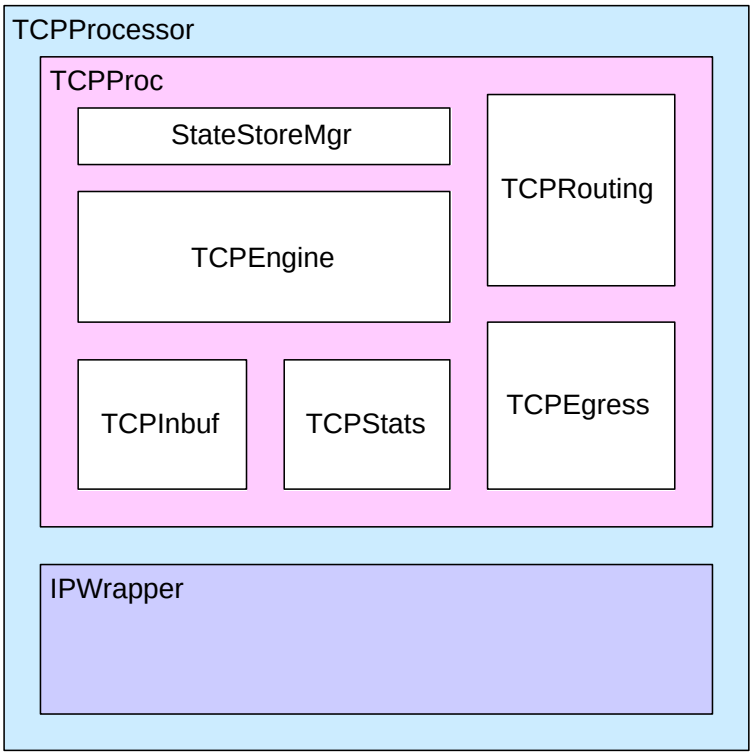


Figure 6.1: Hierarchy of TCP-Processor Components

6.2 Packet Parameters

The TCP-Processor supports packet lengths of 1500 bytes or less. When the circuit is presented with packets larger than 1500 bytes, the viability of the operation of the circuit is not guaranteed. The circuit could likely handle larger packets, but would be subject to lockups when packet lengths exceed the buffering capacity of internal FIFOs.

The TCP-Processor does not support IP fragments. IP defragmentation is a lower layer protocol function which should occur within the IPWrapper. The IPWrapper does not contain any logic to reassemble IP fragments into complete packets, or even validate the IP packet length. The behavior of the circuit is undefined when processing IP fragments.

6.3 Flow Control

Within the confines of the TCP-Processor, it is assumed that whenever the TCA signal is driven low, the sending component can continue to send data until the end of the current packet is reached. This behavior contrasts with how the other protocol wrapper circuits

manage flow control signals. The advantage of this behavior is that it simplifies the logic associated with processing TCA signals. The disadvantage is that it requires larger buffers and/or FIFOs in order to store the additional data after deasserting TCA. This behavior is internal to the TCP-Processor and affects components within the TCP-Processor and any client monitoring application which interfaces with the TCP-Processor. When interfacing with the IP Wrapper, the TCP-Processor conforms to its flow control semantics.

6.4 External Memories

The TCP-Processor utilizes SDRAM memory module 1 to maintain per-flow context information. This task completely consumes this memory module and it should therefore not be used for any other purpose. SDRAM memory module 2 is not used by the TCP-Processor and is available for other features, such as IP packet defragmentation or TCP packet reordering. The two SRAM banks capture packet data for debugging purposes. By eliminating the debugging feature, both of these high speed memory devices could be used for other purposes.

6.5 Configuration Parameters

The TCP-Processor uses several configuration parameters to control its operational behavior. The following list describes each of those parameters in detail, along with possible values and the effect that parameter has on the operation of the TCP-Processor circuit:

num_pre_hdr_wrds- Passed to the IPWrapper component where it is used to determine the start of the IP packet header. Known possible values for this parameter are 0: for ATM environments, 2: for ATM environments with a two word LLC header, 4: for gigabit Ethernet environments, and 5: for gigabit Ethernet environments with a VLAN tag.

simulation- Indicates whether or not the TCP-Processor circuit should operate in simulation mode. When set to 1, it skips memory initialization (the zeroing of external memory). Additionally, it performs all accesses to external SDRAM at address location zero. This parameter enables the quick simulation of network traffic because it avoids the initialization of large external memories. This parameter should be set to 0 when building circuits to operate in hardware.

skip_sequence_gaps- When set to 1, this parameter indicates that the TCPEngine should skip sequence gaps and track the highest sequence number associated with every flow. If a sequence gap occurs, the TCPEngine sets the NEW_FLOW flag to indicate that data in this packet represents a new stream of bytes. The FLOW IDENTIFIER and the FLOW INSTANCE NUMBER remain the same as the previous packet associated with the same flow. This information can be used by the monitoring application to determine the occurrence of a sequence gap. When set to 0, the TCP-Processor drops out-of-sequence packets until packets with the appropriate sequence number arrive.

app_bypass_enabled- Utilized by the TCPRouting module which determines where to route packets. When set to 1, retransmitted packets, non-classified TCP packets (see next configuration parameter), and non-TCP packets are allowed to bypass the client monitoring application and pass directly to the TCPEgress component. When set to 0, all packets are routed to the client monitoring application.

classify_empty_pkts- Indicates whether or not TCP packets which contain no user data should be classified. Performance of the circuit can be improved by avoiding classification of these packets because it takes longer to classify a packet and associate it with a flow than it takes for these small packets to be transmitted on an OC-48 (2.5Gbps) link. When enabled, all TCP packets are classified. This parameter should be enabled if the client monitoring application is tracking the connection setup/teardown sequences or is trying to process bidirectional traffic and match acknowledgment numbers with corresponding sequence numbers of traffic in the opposite direction.

cksum_update_ena- When set to 1, the TCPEgress component validates the TCP checksum for all TCP packets passed in by the client monitoring application. If the TCP checksum value is determined to be incorrect, a correcting suffix is added to the end of the packet which will replace the erroneous checksum value with the proper checksum value. When set to 0, no checksum processing is performed by the TCPEgress component.

stats_enabled- Indicates whether or not a statistics packet should be generated at a periodic time interval. When set to 1, the generation of statistics packets is enabled and conforms to the following configuration parameters. When set to 0, statistics information will not be kept nor transmitted.

stats_id- This 8-bit wide configuration parameter can be used to differentiate statistics generated by multiple TCP-Processors. For example, a certain configuration consists of

three different instances of the TCP-Processor circuit. All three are sending statistics information to the same collection facility. This configuration parameter differentiates the statistics information among the three different TCP-Processor circuits.

stats_cycle_count- This 32-bit wide configuration parameter indicates the interval, in clock cycles, between the sending of a UDP packet containing statistics information collected during the previous collection interval. This parameter is highly dependant on the frequency of the oscillator used to drive the RAD. During simulations, this parameter can be set to a very small number so that statistics packets are generated on a much more frequent basis.

stats_vci- This 8-bit wide configuration parameter specifies the VCI on which the statistics packet should be generated. By altering this value, statistics packets can be routed differently than normal monitored traffic.

stats_dest_addr- This 32-bit wide configuration parameter specifies the destination IP address for the statistics packets.

stats_dest_port- This 16-bit wide configuration parameter specifies the destination UDP port number for the statistics packets.

6.6 TCP Processor

The TCPProcessor component provides the top level interface for the TCP-Processor and is implemented in the `tcpprocessor.vhd` source file. The lower layered protocol wrappers within this component are integrated with the various TCP processing components. Figure 6.2 contains a layout of the TCPProcessor circuit along with groupings of the main signal bundles and how they are connected among external interfaces and internal components.

6.7 TCP Proc

The TCPProc component, implemented in the `tcpproc.vhd` source file, is an integration point for the core components of the TCP-Processor circuit. The TCP-Processor consists of several manageable modules to isolate the various functionalities of the components. It consists of the following six components: TCPInbuf, TCPEngine, StateStoreMgr, TCPRouting, TCPEgress, and TCPStats. Well-defined interfaces allow data to transition among these components. The interface specification will be described in later sections.

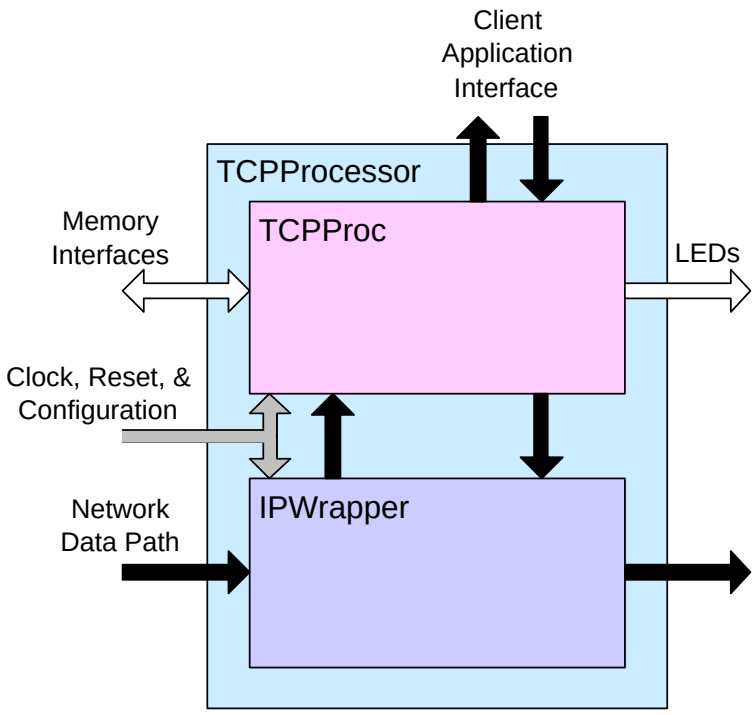


Figure 6.2: TCPProcessor Layout

Figure 6.3 describes a layout of the TCPProc circuit along with some of the main interconnections between components.

The external interface of the TCPProc component connects two SRAM memory interfaces. Inside the TCPProc component, there are three possible connection points for the two SRAM devices: two in the TCPInbuf module and one in the TCPEgress module. To accommodate this difference, the TCPProc maintains a virtual NOMEM1 memory device. It drives selected NOMEM1 signals to give the appearance that the memory device is always available, but writes to the device never take place and reads from the device always result in the value of zero. By changing the connections within the TCPProc circuit, selected subcomponent interfaces can connect to external memory devices and others can connect to the virtual device. This allows for the capture of network traffic at various locations within the circuit which aids in debugging.

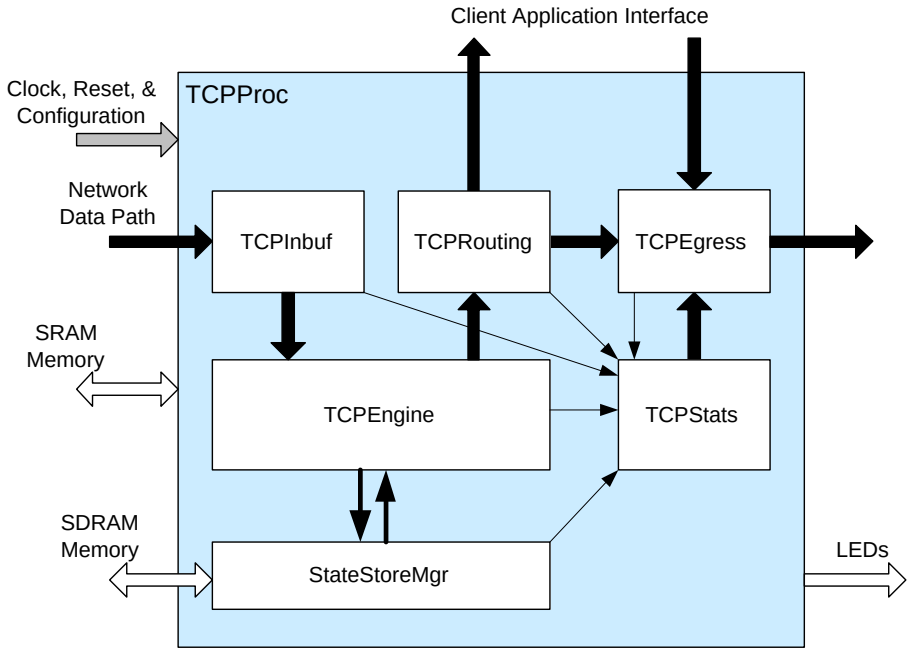


Figure 6.3: TCPProc Layout

6.8 TCP Input Buffer

TCPInbif, implemented in the `tcpinbuf.vhd` source file, is the first component to process network traffic that enters the TCP-Processor. It provides packet buffering services for the TCP-Processor when there are periods of downstream delay. Back pressure induced by downstream components via a flow control signal tells the TCPInbuf component not to release any more packets into the TCP-Processor Engine. In addition, the TCPInbuf component ensures that only fully formed IP packets are passed into the downstream TCP-Processor components. The TCPInbuf component always asserts the upstream flow control signal, indicating that it is always ready to receive data. If its internal buffers are filled to capacity, then TCPInbuf manages the dropping of packets from the network instead of deasserting the TCA flow control signal. Figure 6.4 describes the layout of the TCPInbuf component.

As network traffic enters the TCPInbuf component, it is first processed by the input state machine. The state machine determines whether the packet should be dropped (because the Frame FIFO is full) or should be inserted into the Frame FIFO. A separate packet length counting circuit counts the number of 32-bit words associated with the packet which are inserted into the Frame FIFO. At the end of the packet, this length value is written to the

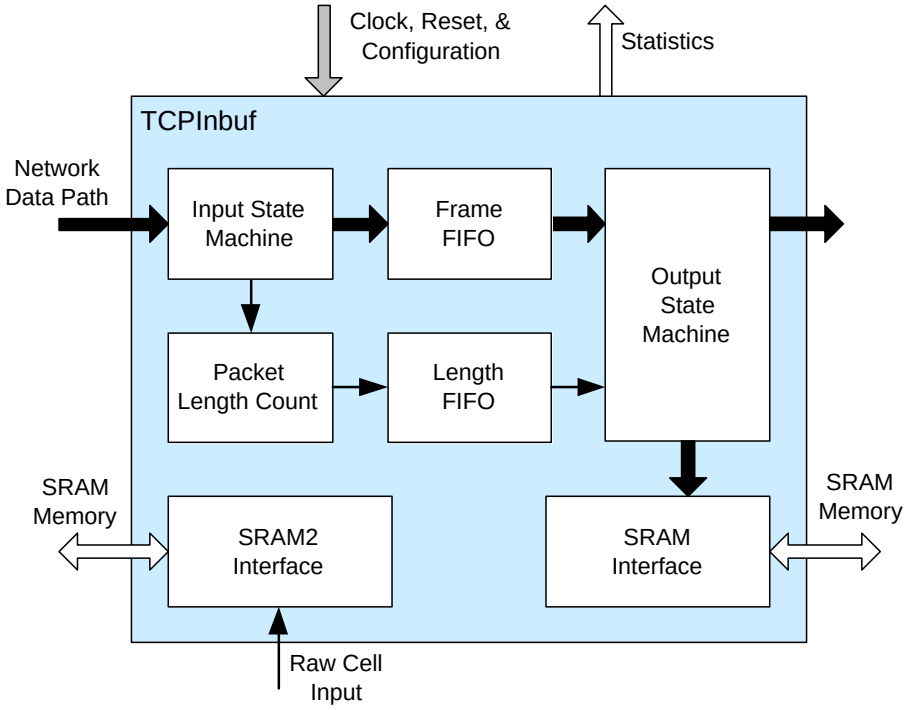


Figure 6.4: TCPInbuf Layout

Length FIFO. If only a portion of the packet was successfully written to the Frame FIFO (because the FIFO filled up while inserting words), then only the length associated with the number of words actually inserted into the Frame FIFO is passed to the Length FIFO.

The output state machine utilizes one of two different methods to retrieve data from the FIFOs and forward it downstream: uncounted mode and counted mode. It uses the uncounted mode when there are no packets queued in the Frame FIFO. This mode does not induce and store and forward delay for the packet while the packet length is being computed, which is a performance benefit. When the output state machine detects a non-empty Frame FIFO and empty Length FIFO, it enters the uncounted processing mode. Packet data words are clocked out of the Frame FIFO and passed to the output interface of the TCPInbuf component. When TCPInbuf reads the final word of the packet from the Frame FIFO, it then reads the length of the packet from the Length FIFO.

The output state machine enters the counted mode after there has been some down-stream delay in the system and one or more packets are queued up in the Frame and Length FIFOs. It also enters this mode when it is in the idle state and data is present in both the Length and Frame FIFOs. It reads the packet length first from the Length FIFO. Bits 8 through 0 indicate the length in units of 32-bit words. Bit 9 is a special drop flag which

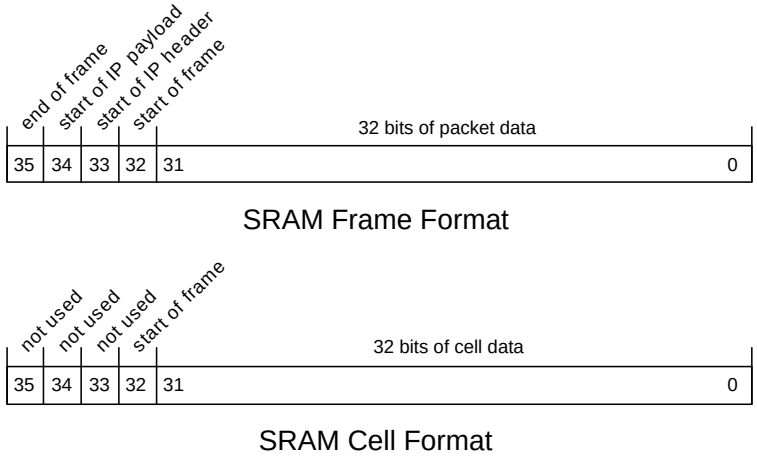


Figure 6.5: SRAM data Formats

indicates whether or not the packet should be dropped. This flag is set whenever an incomplete packet is inserted into the Frame FIFO. If the drop flag is set, then the specified number of words is read from the Frame FIFO and discarded. If the drop flag is zero, the the specified number of words is read from the Frame FIFO and passed to the output interface of the TCPInbuf component.

The packet storage and SRAM signals manage two separate SRAM devices used to capture packet data for debugging purposes. After reset, both SRAM memory devices are initialized to zero. The control circuits are configured to use locations 1 through 262,143 as a ring buffer. Packets passed to the outbound interface are written to the first SRAM interface and inbound ATM cells from the external RAD interface are written to the second SRAM interface. When the termination criteria has been met, the memory update logic is put into a mode where it stores the current memory write address to location zero. In this manner, the 262,143 32-bit packet data words can be captured prior to the detected event. Packets (frames) and cells are stored in the standard format required by the SRAMDUMP utility. Figure 6.5 illustrates the layout of the 36-bit SRAM memory for both of these formats.

6.9 TCP Engine

The TCPEngine component is the workhorse component of the TCP-Processor and is implemented in the `tcpengine.vhd` source file. It contains all of the complex protocol processing logic. Figure 6.6 shows a breakdown of the major logic blocks and data flow

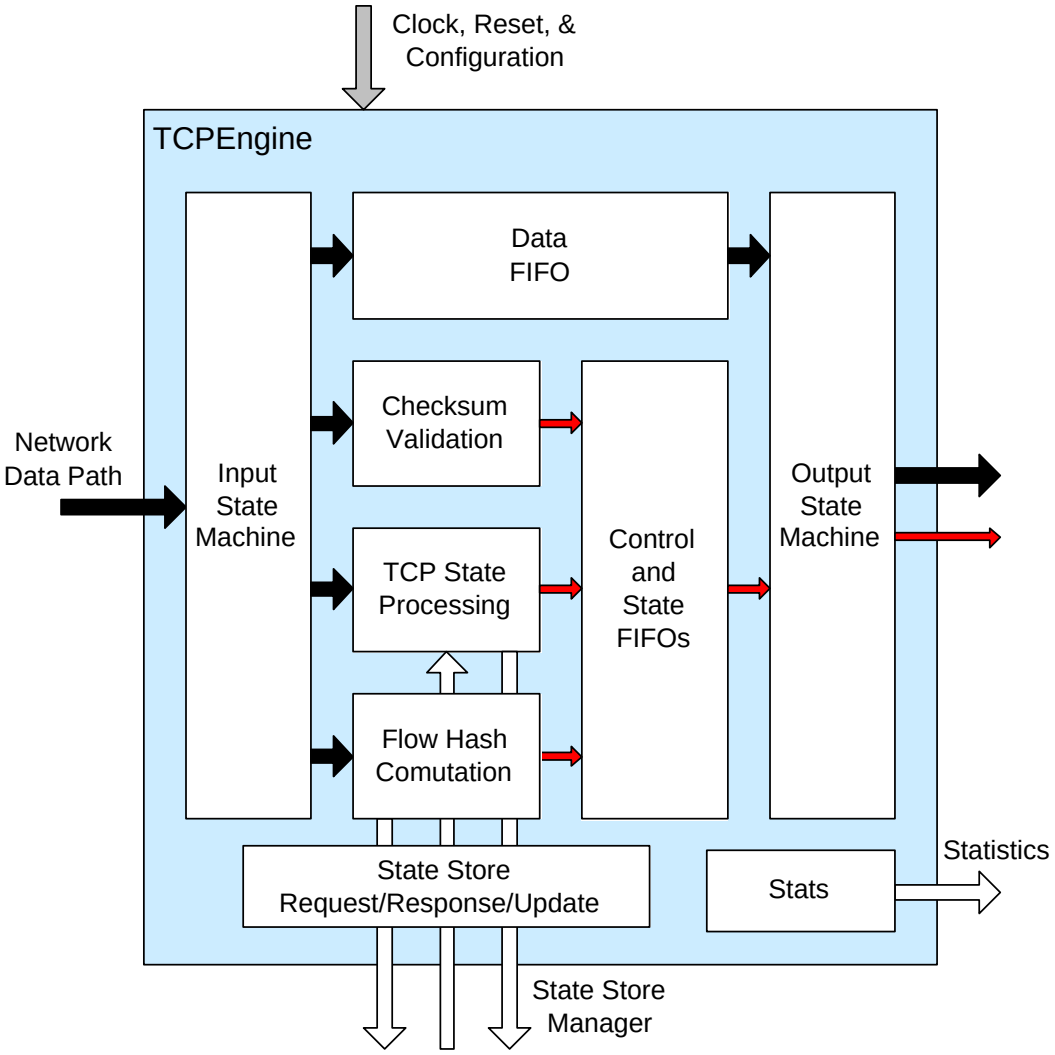


Figure 6.6: TCPEngine Layout

through the component. Network traffic travels left to right. Flow classification and interactions with the state store manager are handled at the bottom of the diagram. The input state machine and output state machine are noted by separate boxes at the left and right sides of the figure. Additional state machines manage interactions with the state store manager. The state store request/response/update block of logic contain these interactions.

The TCPEngine has evolved over time and contains remnants of logic for features, such as the direction signal and multiple hash computations, which are no longer utilized. VHDL code for these features remains for possible future implementation. Since this document describes a version of the TCP-Processor which contains this unused logic, its presence is noted.

The first series of processes described in the `tcpengine.vhd` source file generate event signals which are passed to the TCPStats component. These signals pulse for one clock cycle to indicate when the associated event counter should be incremented.

The input state machine is the first component to process IP packets when they enter the TCPEngine. It tracks the main processing state for the TCPEngine component while processing inbound packet data. The individual states refer to the part of the packet being processed. IP header processing states include version, identifier, protocol, source IP, destination IP, and option fields. TCP header processing states include ports, sequence number, acknowledgment number, data offset, checksum, and option fields. Data0 and data1 states represent processing of TCP payload data. The circuit enters the copy state when it encounters a non-TCP packet and copies the packet to the Frame FIFO. The length and CRC states refer to the AAL5 trailer fields which are tacked on to the end of the packet. The circuit enters the WFR state when it is waiting for a response from the state store manager.

The next section of logic deals with extracting protocol-specific information from the protocol headers. This information includes the IP header length, an indication of whether or not the protocol is TCP, the TCP header length, the current header length, the value of various TCP flags, the source IP address, the destination IP address, and the TCP port numbers.

The TCPEngine next computes a hash value based on the source IP address, the destination IP address, and the source and destination TCP ports. Many different hashing algorithms have been used throughout the development of the TCP-Processor and source code still exists for several of them. The current hash scheme is hash3 which combines the results of a CRC calculation over the various input values. The hash algorithm is constructed so packets in the forward and reverse directions hash to the same value. This is accomplished by XORing the source fields with the destination fields in an identical manner. The resultant hash value is 23 bits wide, which corresponds to 2^{23} or 8,388,608 unique hash values. This hash value is passed to the state store manager which performs the flow classification operation.

The next section of logic deals with operations required to initiate a flow lookup operation via the state store manager. A request state machine handles the required state transitions. After initiating the request, the state machine enters a hibernate state waiting for the end of input processing for the current packet. This ensures that multiple state store manager lookup requests are not made for one single packet. Table 6.1 lists the sequence

of data words the state machine passes to the state store manager when initiating a request for per-flow context information.

Table 6.1: State Store Request Sequence

Request word	Request data (32 bit field)
1	”000000000” & computed hash value
2	source IP address
3	destination IP address
4	source TCP port & destination TCP port

The following section contains logic to manage the per-flow context data returning from the state store manager. A response state machine tracks this response data. There are three paths that that the context data can traverse through the state machine. It enters the first when the state store manager returns a new flow context record. The second path corresponds to a valid flow context lookup of an existing flow, and the context data enters the third path when a per-flow context lookup is not performed, requiring a place holder. Table 6.2 shows the state transitions along with the data returned from the state store manager. Bit 31 of the first response word indicates whether or not the response corresponds to a new flow (value 0) or an existing flow (value 1). The returned flow identifier is a 26-bit value. The low two bits of the flow identifier (bits 1-0) are always zero. Utilizing the whole flow identifier as a memory address for the SDRAM on the FPX platform provides access to a separate 32-byte section of memory for each unique flow. Bit 2 of the flow identifier is a direction bit. If unidirectional traffic alone is being monitored (i.e., traffic propagating the network in a single direction), then the value of this bit will always be zero. This also implies that 64 bytes of data can be stored for each unidirectional flow. If bidirectional traffic is being monitored, then the flow identifiers of outbound and inbound traffic associated with the same TCP connection will differ only by the value of this bit. There is no correlation between the value of this bit and the direction of the traffic. The only conclusion that can be drawn is that packets with flow identifiers that have the bit set are traversing the network in the opposite direction of packets with flow identifiers that have the bit cleared.

The next section of logic contains processes which perform TCP specific protocol processing functions for the TCP-Processor. This includes a process which loads state/context information into the app state FIFO. When the state machine enters the NIL

Table 6.2: State Store Response Sequence

	New flow		Existing flow	
Response word	State	Response data (32 bit field)	State	Response data (32 bit field)
1	Idle	”000000” & flow id	Idle	”100000” & flow id
2	Instance	x”000000” & instance	Sequence	current sequence num
3			Instance	x”000000” & instance

states, zero values are written into this FIFO instead of valid data. This occurs when processing non-TCP packets. The contents of this FIFO will eventually be passed to the client monitoring application via the flowstate signal lines. Four data words are written to this FIFO for every packet processed. The data words either contain zeros or valid context information, depending on the type of packet being processed and the TCP-Processor configuration settings. Other logic in this section of code computes a byte offset to the first byte of new data for the flow and computation of the next sequence expected sequence number.

Logic also exists to send updated per-flow context information to the state store manager. The updated context information includes the next expected sequence number for the flow and an indication of whether or not the connection is being terminated. A signal can also be passed to the state store manager indicating that an update is not being generated and the state store manager should return to its request processing state. Additional signals indicate whether or not this packet is associated with a new flow, keep track of the flow identifier, and compute the length of the TCP payload. The checksum calculation logic, next in the source file, selects the appropriate words of the IP header, TCP header, and TCP payload to be used in the TCP checksum calculation.

The following section of logic writes control information into the control FIFO. The control FIFO contains information relating to the validity of the TCP checksum, an indication of whether or not this is a TCP packet, the offset to the next byte in the TCP data stream, the number of bytes in the last data word of the packet, a new flow indication, an indication of whether or not the sequence number is in the proper range, an indication of whether or not the packet should just be forwarded (i.e., for retransmitted packets), and an end-of-flow indication. Figure 6.7 provides a layout of this data.

The next process is associated with writing packet data into the data FIFO. The data inserted into this FIFO includes 32 bits of packet data, a start of frame indication, a start of IP header indication, an end of frame indication, a start of IP payload (start of TCP header)

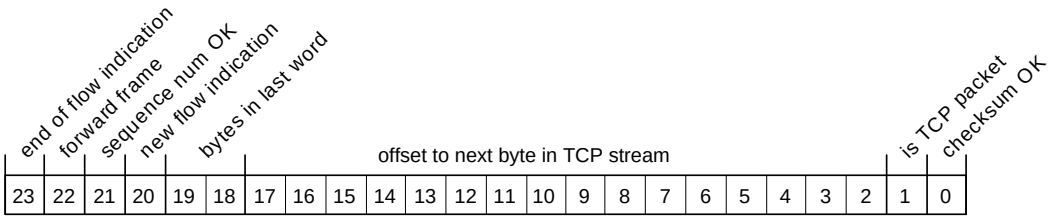


Figure 6.7: Control FIFO data format

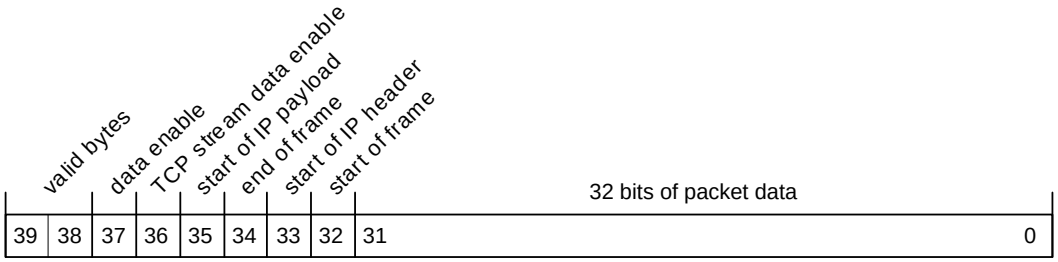


Figure 6.8: Data FIFO data format

indication, a TCP stream data enable signal, a regular data enable signal, and a valid bytes vector indicating how many bytes contain stream data. The valid bytes vector is offset by one, so the value ”00” indicates one valid byte, ”01” indicates two valid bytes, ”10” indicates three valid bytes, and ”11” indicates four valid bytes.

The remainder of the TCPEngine VHDL code retrieves data out of the three FIFOs, performs some final data modifications, and passes the information out of the TCPEngine component and onto the TCPRouting component. This whole process is driven by the output state machine which, upon detecting a non-empty control FIFO and app state FIFO, initiates the sequence of events to read information from the FIFOs and pass it to the external interface. It sends control signals to the control, data, and app state FIFOs to start extracting their contents. As data is retrieved from the data FIFO, computations are carried out to generate the valid_bytes signal which indicates the number of valid TCP bytes (zero, one, two, three, or four) contained on the packet data bus. Data read from the app state FIFO is passed to the flowstate vector. An offset mask and a length mask are added to this set of data which specify a mask of valid bytes for the first and last words of the stream data that should be considered part of the TCP data stream by the client monitoring application. The offset signals are used to determine where in the packet the client monitoring application should start processing TCP stream data.

6.10 State Store Manager

The StateStoreMgr component provides simple interfaces for accessing and updating per-flow context information stored in external SDRAM and is implemented in the `statestoremgr.vhd` source file. The StateStoreMgr provides two separate interfaces for accessing per-flow context information. The first is used by the TCPEngine, presenting a per-flow hash value along with the source IP, destination IP, and source and destination TCP ports. The StateStoreMgr navigates to the existing context record for that flow, or allocates new memory to hold context information for the flow. The second interface is unused. It was originally developed to provide the back end of the TCP-Processor access to per-flow context information and supported flow blocking and application state storage services. This second interface is not utilized in this version of the TCP-Processor, but much of the logic remains in place for possible future use.

A version of the TCP-Processor which supports flow blocking and application state storage can be found at the following URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/source/streamcapture_source.zip`. Please note that the StreamCapture version of the TCP-Processor is an older version of the source.

The TCPEngine is highly extensible due to the fact that the logic which manages the per-flow context memory is separate, allowing for the creation of different lookup, retrieval and memory management algorithms without requiring complex changes to the TCPEngine. The SDRAM memory controller[1] utilized by the StateStoreMgr provides three separate interfaces: a Read only interface, a Write only interface and a Read/Write interface. Figure 6.9 shows the layout of the StateStoreMgr state machines and their interaction with other components. As previously stated, the second request/update interface is unused. The data bus to the SDRAM controller is 64 bits wide and the data bus exposed by the StateStoreMgr is 32 bits wide. In order to handle the transition between the two different bus widths, a separate 8x64 dual-ported RAM block is utilized in each of the interfaces.

The first process of the `statestoremgr.vhd` source file generates pulses reflecting internal events which are passed to that TCPStats module. The TCPStats module maintains statistical counters and events including the occurrence of a new connection, the occurrence of a reused connection (i.e., an active flow context record has been reclaimed

[1] developed by Sarang Dharmapurikar

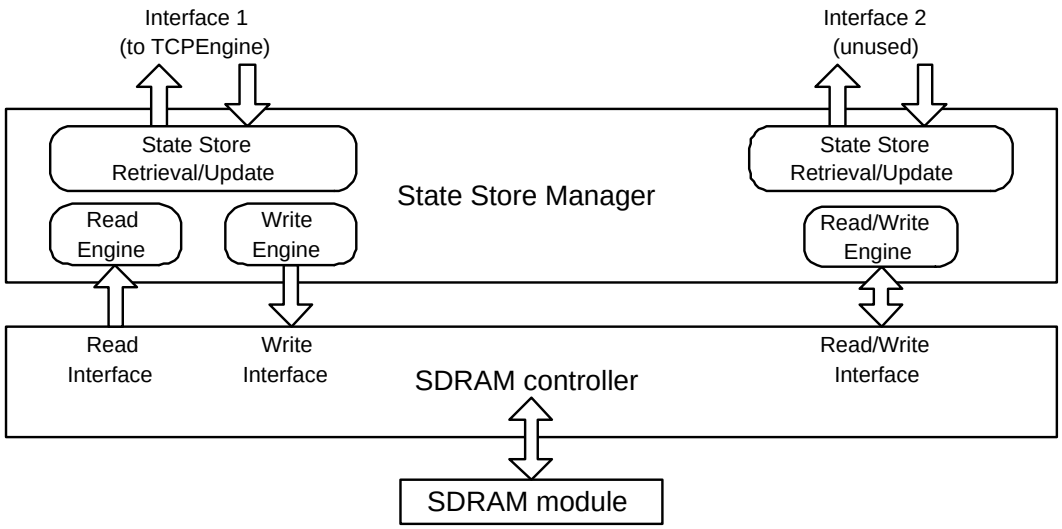


Figure 6.9: Layout of StateStoreMgr

and is being used to keep state information for a different flow), the occurrence of a terminated flow (i.e., an existing TCP connection has been closed and no longer requires state storage resources).

Two major sections make up the remainder of the StateStoreManager logic. The first section handles frontside processing associated with interface 1 and the second sections handles backside processing associated with interface 2. Interface 1 is connected to the TCPEngine and performs the flow classification, context storage and retrieval operations required by the TCP-Processor. Interface 2 is not used in this instance, but contains logic for performing flow blocking and application state storage services. Enhancements have been made to Interface 1 which have not been incorporated into Interface 2. Before attempting to utilize Interface 2, the inconsistencies between the two interfaces will have to be resolved.

The operations of interface 1 are driven by the request1 state machine. Upon detecting a request, the state machine traverses through states to save the additional request data. It enters a delay state waiting for a memory read operation to complete. Once data returns from external memory, the state machine's sequence of states return results to the requestor. An additional state handles the processing of a context update message. Some states contained within this state machine, which support the retrieval of client monitoring application context information not supported in this version of the TCP-Processor, are never reached.

The next four processes capture flags are stored in SDRAM. Two of these flags indicate whether or not the inbound and outbound traffic is valid. The other two flags indicate whether or not the inbound or outbound flow has been terminated. These four flags help the StateStoreMgr to determine the operational state of the context record of a particular TCP flow (i.e., whether or not the record is idle, active for inbound traffic, active for outbound traffic, or whether traffic in either direction has been shut down). It should be noted that the inbound and outbound traffic directions do not correspond to the specific direction of the traffic, rather that inbound traffic is moving in the opposite direction of outbound traffic. Across multiple TCP flow context records, the inbound traffic in each of these flows may actually be moving through the network in different directions.

The subsequent two processes manage the exact flow context matches between the context record stored in external memory and the four-tupple of source IP address, destination IP address, and source and destination TCP ports passed in by the requestor. The match1 circuit performs the match for outbound traffic and the match2 circuit performs the match for inbound traffic (i.e., the match2 circuit swaps the source and destination parameters).

The following processes store individual data items from the context record returned from external memory. They store the current sequence numbers for both outbound and inbound traffic along with the instance identifier for that flow. As the per-flow context data clocks in from memory, these data values are pulled off of the SDRAM read data bus. The instance identifers are maintained for each flow record. Every time a different TCP connection associates with a flow record, the instance identifier of that record increments. This behavior provides a simple mechanism for downstream stream monitoring clients to quickly know when a flow record is reclaimed and a different TCP connection is mapped to the same flow identifier (i.e., memory record).

The next set of processes stores data items passed in as part of the initial request or computed along the way. The flow hash value, one of these data items, is used as the initial index into the state store manager hash table. The direction flop determines the direction of the flow based on which of the two flow matching processes returns a successful match. The upd_direction stores the packet direction during update processing. This allows the request engine to start processing another lookup request. As the source IP addresses, destination IP addresses, and the TCP port values pass in from the requestor, they are saved and used in later comparisons by the match processes.

The response1 engine generates a response to the requesting circuit. Based on the current processing state, results of the match operations, and the value of the valid flags.

The response sequence differs depending on whether or not there was a successful per-flow record match. If an existing flow record is not found, the response1 engine returns a flow identifier and an instance number. If an existing flow record is found, the response1 engine also returns the current sequence number. Table 6.2 shows the returned data in both of these scenarios.

A read state machine manages the interactions with the SDRAM controller read interface. It sequences through PullWait, TakeWait and Take states as defined by the memory controller (developed by Sarang Dharmapurikar). Prepare and Check states ensure that the frontside and backside engines are not trying to operate on the same per-flow record at the same time. Since the backside engine is not used in this incarnation of the TCP-Processor, these states will never be entered. The read engine uses the aforementioned states and drives the SDRAM controller signals to affect a read operation.

Figure 6.10 illustrates the memory layout of a bidirectional per-flow context record. The structure contains many unused bytes which can be utilized in conjunction with future circuit enhancements. The source IP address, the destination IP address and the source and destination TCP ports are stored within the per-flow record so that (1) an exact match can be performed to ensure that the per-flow record corresponds to the flow being retrieved and (2) the direction of the traffic (either outbound or inbound) can be determined. The per-flow record stores the current sequence number for both outbound and inbound traffic and an 8-bit instance identifier. This instance value increments each time a new flow is stored in a context record. Four 1-bit flags indicate whether or not outbound traffic is valid, inbound traffic is valid, outbound traffic has ended and inbound traffic has ended.

The next couple of processes indicate whether or not the frontside engine is active and if active, reveals the memory address of the per-flow record being accessed. When both the frontside and backside engines are utilized, these signals ensure that both engines don't try to manipulate the same context record, potentially leaving the record in an inconsistent state. In addition, three counters keep track of the take count, the app count, and the update count. The take count refers to the data words returned from SDRAM. The app count refers to the application-specific context information stored in the context record. The storage of application-specific context information is not utilized in this version of the TCP-Processor. The update count keeps track of memory update processing.

The SDRAM controller utilized by the StateStoreMgr contains a 64-bit wide data bus. The external request/response interface uses a 32-bit wide data bus. An internal dual-ported memory block moves data between these two busses. This memory block is called frontside_ram. Two engines control the movement of data in and out of the memory. As

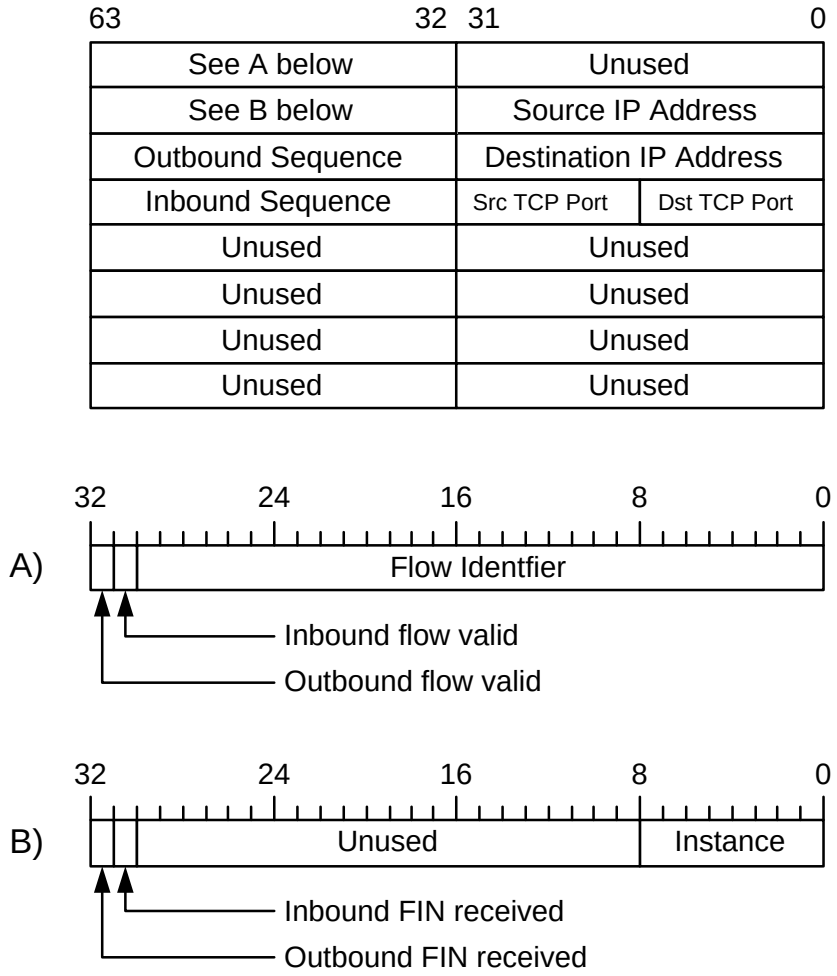


Figure 6.10: Per-Flow Record Layout

data is clocked in from external memory, the frontside_ram_A_engine writes this data into the frontside_ram. This data is written 64 bits at a time which matches the width of the memory interface. The frontside_ram_A_engine also reads data out of the frontside_ram where it is written to the external memory device. If the application context storage feature were enabled, then read operations utilizing the frontside_ram_B_engine would read application data from the frontside_ram to be used in the response sequence to the TCPEngine. The logic still exists for this task to support possible future enhancements which would require additional context storage. Currently, the state which drives this operation is never entered. Updates to the per-flow context record are written to the frontside_ram utilizing the frontside_ram_B_engine. Figure 6.11 shows the various interactions with the frontside_ram.

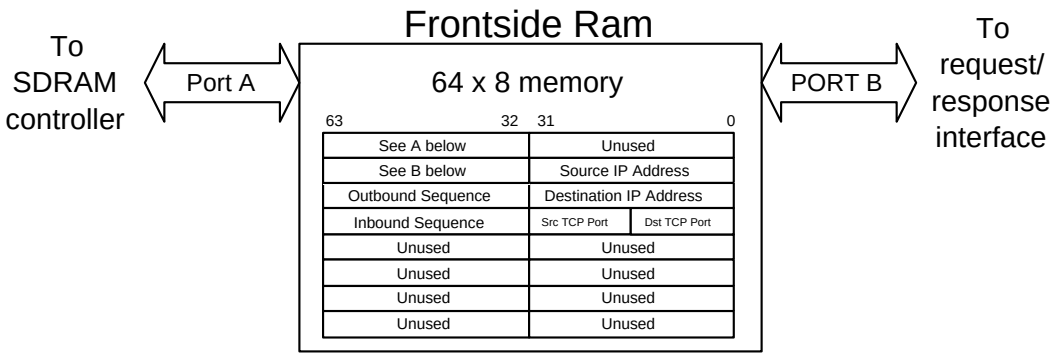


Figure 6.11: Frontside RAM Interface Connections

The write state machine manages interactions with the write interface of the memory controller. This state machine also initializes memory after reset. The write state machine enters a series of states which request the memory bus and write zeros to all memory locations of the external memory module. This initializes memory to a known state prior to any per-flow context record storage or retrieval operations. The write engine ensures the correct interaction with the write interface of the memory controller.

The next group of processes increments a memory address utilized during memory initialization and a flag to indicate that initialization has completed. Additionally, there are signals to hold the update information passed in via the external interface. This information includes the new sequence number and a finish signal which indicates that the TCP connection is being terminated. The final process of the frontside section maintains a counter utilized when writing data to external memory.

The remainder of the source file consists of a list of processes associated with the backside request/response interface. The actions these processes take closely mirror the actions taken by the frontside interface. Because of this similarity, and the fact that the backside interface is not used in this implementation of the TCP-Processor, the details of the operation of these processes and signals will not be presented.

6.11 TCP Routing

The TCPRouting component routes the packets and associated information that arrive from the TCPEngine component and are implemented in the `tcprouting.vhd` source file. These packets can be routed to either the client monitoring interface (the normal traffic

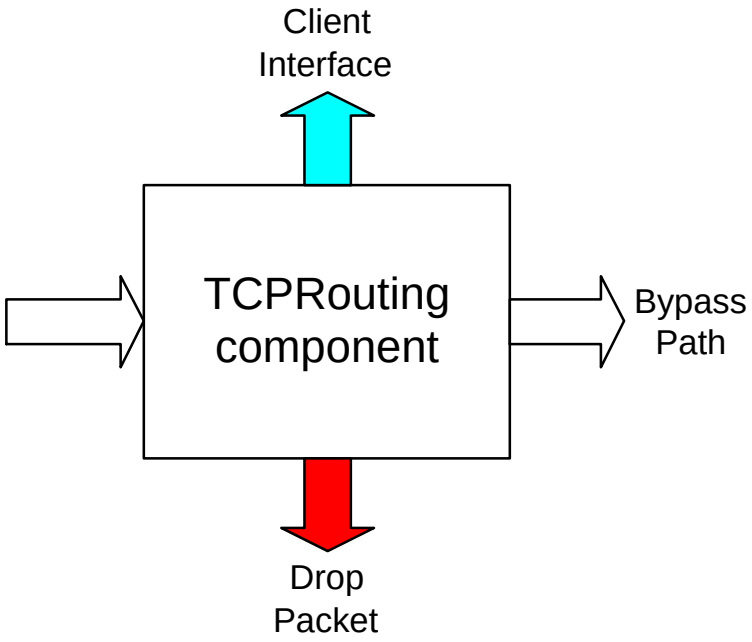


Figure 6.12: TCPRouting Component

flow), the TCPEgress component (the bypass path), or the bit bucket (i.e., dropped from the network). Figure 6.12 shows the layout of the TCPRouting component.

The first process in the `tcprouting.vhd` source file generates the statistics event signals passed to the TCPStats component. These signals keep track of the number of packets passed to each of the output interfaces. Additionally, a two-bit vector keeps an accurate count of the TCP stream bytes passed to the application interface. This count does not include retransmitted data. The next group of processes contains flops which copy all inbound data to internal signals. These internal signals are used when passing data to the output interfaces. One state machine in this component tracks the start and the end of a packet. The processing states include idle, data, length, and crc states. The final two states indicate the end of an IP packet. Since the TCPEngine contains a store and forward cycle, packets are always received by the TCPRouting component as a continuous stream of data.

The next two processes determine whether or not the TCPRouting component enables the client application interface, the bypass interface, or neither interface (i.e., drop packet). Information passed in from the TCPEngine and the current configuration setting are used to make this decision. The TCPRouting component generates two enable signals used by subsequent processes which actually drive the output signals.

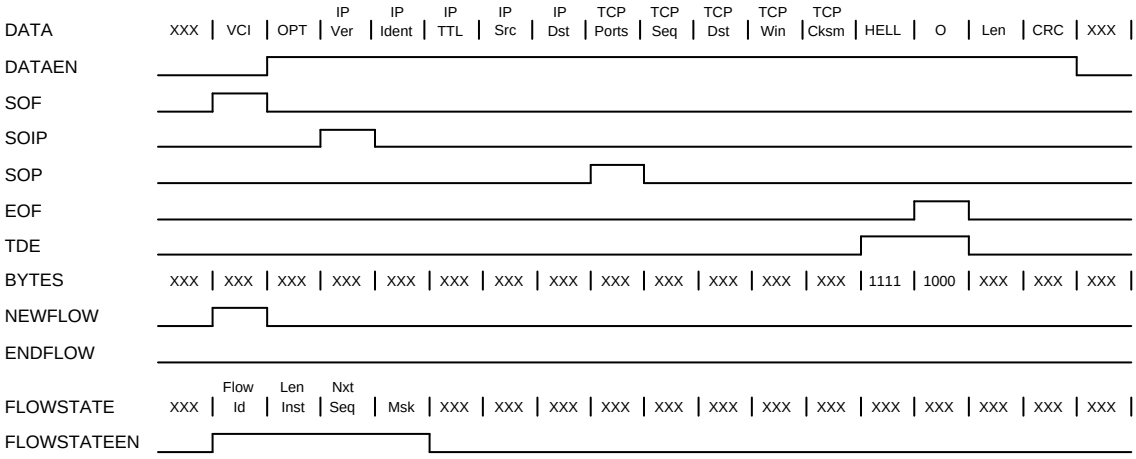


Figure 6.13: TCP Outbound Client Interface

The remainder of the TCPRouting processes drive output signals to either the client application interface or the bypass interface. These processes look at the current processing state and the value of the aforementioned enable signals to determine whether or not to connect the internal data signals to the output interfaces.

6.11.1 Client Interface

Figure 6.13 illustrates a waveform showing the interface signals from the TCP-Processor to the client TCP flow monitoring application. The interface consists of 12 output signals (data, dataen, sof, soip, sop, eof, tde, bytes, newflow, endflow, flowstate, and flowstaten) and one input signal (tca) used for flow control purposes. A high sof signal initiates packet data processing. In conjunction with this signal, the newflow and/or endflow signals indicate whether this packet is the start of a new flow or the end of an existing flow. It is possible for both the newflow and endflow signals to be high for the same packet. This can occur if the first packet processed for a particular flow has the RST or FIN flags set. The flowstate and flowstaten signals are also driven in conjunction with the sof signal. The data bus contains the 32-bit ATM header (minus the HEC) associated with the packet. The dataen signal is low for the first word of the packet, but is driven high for all successive data words.

Following the initial clock cycle of the packet, zero or more lower layer protocol header words inserted prior to the start of the IP header may exist. In this example, there is one such header word labelled OPT. The soip signal indicates the start of the IP header.

Only IPv4 packets are supported, which means there will be at least 20 bytes of IP header data. If IP header options are present, they occur prior to the sop signal. The sop signal indicates the start of the IP payload. In this example, the sop signal corresponds to the start of the TCP header. The TCP header is 20 bytes or larger, depending on the number of included TCP options.

The packet in Figure 6.13 contains the five character text string *HELLO* in the TCP payload section of the packet. The client application processing the TCP stream content utilizes the tde and bytes signals to determine which data bytes to process as stream content. The tde (TCP data enable) signal indicates that the information on the data bus is TCP stream data. The four-bit-wide bytes vector indicates which of the four data bytes are considered part of the TCP data stream for that flow. When retransmissions occur, it is possible that the bytes vector will indicate that none of the data bytes pertain to TCP stream data.

The eof signal indicates the end of the packet. Upon detection of a high eof signal, two subsequent trailer words containing AAL5 length and CRC information associated with the current packet need to be processed.

Every TCP packet provides four words of flowstate data. Figure 6.14 illustrates the exact format of these data bytes. The flow identifier is always the first word of flowstate information. Bit 31 indicates that the flow identifier is valid. The next data word contains the TCP data length in bytes, the TCP header length in words, and a flow instance number that can be used to help differentiate multiple flows which map to the same flow identifier. The next word contains the next expected TCP sequence number for this flow. This value can be useful when performing flow blocking operations. The final word contains a word offset to the continuation of TCP stream data with respect to this flow along with byte masks for the first and last words of the flow.

6.12 TCP Egress

The TCPEgress component, implemented in the `tcpegress.vhd` source file, merges traffic from different sources into one unified packet stream. This packet stream is passed to the outbound IP wrappers for lower layer protocol processing. Three separate FIFOs buffer traffic from the different sources during the merge operation. The TCPEgress component performs additional processing on the inbound traffic from the TCP flow monitoring client to validate and possibly update TCP checksum values. By recomputing the TCP checksum, the TCP-Processor supports client TCP processing circuits which modify TCP stream

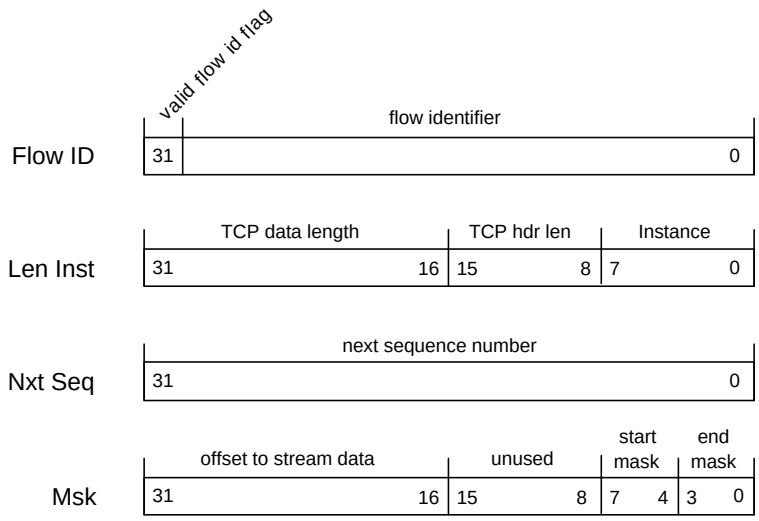


Figure 6.14: Flowstate Information

content. This can be accomplished by either inserting new packets (possibly a packet to terminate the TCP flow), or by modifying an existing packet. The IPWrapper supports special trailer words which indicate modifications that need to be made to the packet. The TCPEgress component does not take these changes into consideration when performing its checksum calculation. If the client uses this mechanism to modify the packet (either header or body), the calculated checksum value will be incorrect and the packet will eventually be dropped. To operate properly, the TCPEgress component must receive properly formatted TCP/IP packets (excluding the checksum values).

Like the previous components, the `tcpegress.vhd` source file begins with a process that generates statistics events passed to the TCPStats component. Statistics are kept on the number of packets processed from the client interface, the number of packets processed from the bypass interface, and the number of TCP checksum update operations performed.

The next set of processes in the source file copies input signals from the statistics and bypass interfaces and flops them into a set of internal signals. These internal signals are then used to write the packet contents to the stats and bypass FIFOs. The stats, app and bypass FIFOs all contain packet data in the same format. Figure 6.16 illustrates this format. A set of processes which clocks inbound packets from the client interface through five separate signals creates a five cycle delay which is utilized later to make checksum results available before the end of the packet is reached. The logic is straightforward and involves no additional processing on the packets.

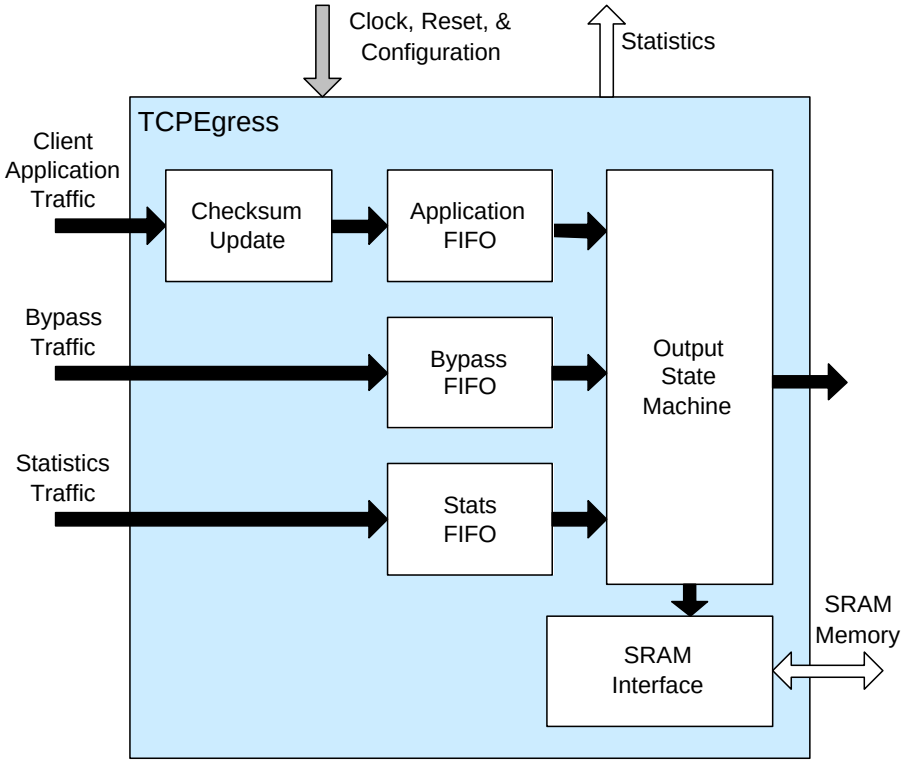


Figure 6.15: TCPEgress Component

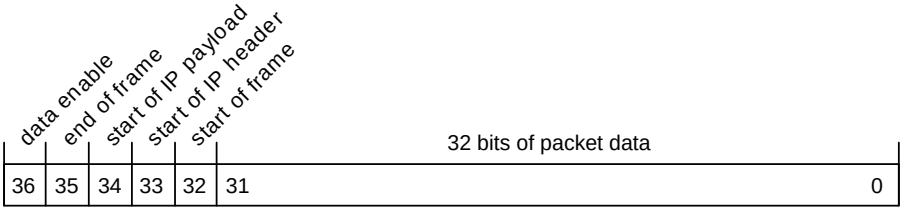


Figure 6.16: TCPEgress FIFO Format

The next section of logic performs the TCP checksum validation operation. An input state machine tracks progress through the various parts of the IP packet. This state machine has similar states and state transitions to the input state machine contained within the TCPEngine component (which also performs a TCP checksum validation operation). Signals compute the length of the IP header and the data length of the TCP packet. This information is used to properly navigate the packet and compute the checksum value.

The set of processes which actually performs the checksum calculation acts separately on the upper and lower 16 bits of the packet since data is clocked 32 bits at a

time. The results of these separate checksum calculations are combined to produce the final checksum result. Following the checksum calculations, a series of processes holds the new checksum value, a determination as to whether the packet checksum is valid, the offset to the checksum value, and the checksum value contained in the packet.

The next group of processes clocks packets from the client application interface into the application FIFO. A simple state machine controls the sequence of events and determines whether the recomputed checksum value should be added to the end of the packet as an update command to the IPWrapper. If a checksum update is required, the state machine enters the Cksum and Delay states which supports inserting the update command into the packet trailer.

The next section of code contains the output state machine which pulls data out of the three internal FIFOs and passes packets to the outbound IPWrapper interface. The state machine prioritizes the traffic in the three FIFOs so packets are always retrieved from the stats FIFO first, the bypass FIFO second, and the app FIFO third. The last_read signal ensures that the output state machine reads a complete packet from the various FIFOs. Timing issues can arise when the state machine attempts to read the last word of a packet from a FIFO and the FIFO is empty because of the delay between when the read command is issued and when the data is returned from the FIFO. The last_read signal helps the processes to prevent deadlocks.

The final set of TCPEgress processes controls interactions with the SRAM memory module. It writes outbound packets to the SRAM device until it fills up with 256k words or packet data. The SRAMDUMP utility retrieves this packet data from memory. Packets are stored in the standard frame format as shown in Figure 6.5 on page 82. All of SRAM is initialized to zero after receiving a reset command. The TCPEgress component will not start processing packets until the memory initialization is complete. Utilizing the *simulation* configuration parameter allows simulations to run quickly by skipping the memory initialization operation.

Packets passed from the client monitoring application to the TCP-Processor must be formatted in the same manner as packets passed to the client monitoring application. The major difference between the format of packets passed to the client and the format of packets passed back from the client is that not all of the control signals are required for packets returned from the client. As seen in Figure 6.17, only the data, dataen, sof, soip, sop and eof signals are necessary. Other than dropping the remaining signals, the interface specification is identical to that of packets delivered through the outbound client interface of the TCP-Processor.

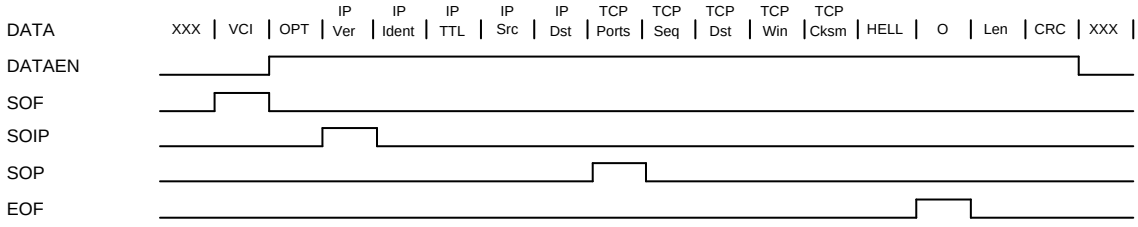


Figure 6.17: TCP Inbound Client Interface

6.13 TCP Stats

The TCPStats component collects statistical information from other components, maintains internal event counters, and generates UDP-based packets on a periodic basis containing a summary of the event counts during the previous collection period. This logic is implemented in the `tcpstats.vhd` source file.

The first two processes in the `tcpstats.vhd` source file maintain a collection interval triggers and a cycle counter. The counter is a free running clock which resets whenever the the interval period (defined by configuration parameter) is reached. Composed of four separate 8-bit counters, the counter eliminates the need for a single 32-bit-wide carry chain. The trigger fires when these four counter values match the state_cycle_counter configuration parameter.

The next group of TCPStats processes contains 16, 24 and 32-bit event counters. It maintains a separate counter for each statistics variable maintained by the TCPStats component. When the trigger fires, these counters are reset, indicating the end of the collection period.

The packet state machine cycles through the processing states required to generate a UDP packet containing a summary of the statistics information gathered over the previous collection period. The trigger signal initiates the generation of the statistics packet. Once started, the process cannot be stopped and the state machine traverses through all of the states required to send the packet.

The next process stores the value of each statistic counter in a temporary holding location while the statistics packet is being generated. When the trigger fires, signalling the end of the collection period, all of the current counter values are saved before being reset to zero. This provides a consistent snapshot of all the counters in a single clock cycle.

The next process generates the outbound UDP-based statistics packets. Information on the packet formats and data collection tools can be found at the following URL:

`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/statscollector.html`

Figure 6.18 shows the processing states and the associated packet information relating to the statistics packet. To add additional statistics to this packet, the developer will has to modify the UDP packet length (the high 16 bits of the Cksum word), the stats count (the low 8 bits of the Hdr word), and insert the new statistics parameters at the end of the current parameters.

The exact sequence of statistics values is listed below:

- cfg1: simulation
- cfg2: application bypass enabled
- cgf3: classify empty packets
- cfg4: checksum updated enabled
- cfg5: skip sequence gaps
- ssm1: new connections
- ssm2: end (terminated) connections
- ssm3: reused connections
- ssm4: active connections
- inb1: inbound words
- inb2: inbound packets
- inb3: outbound packets
- inb4: dropped packets
- eng1: inbound packets
- eng2: TCP packets
- eng3: TCP SYN packets
- eng4: TCP FIN packets
- eng5: TCP RST packets
- eng6: zero length TCP packets
- eng7: retransmitted TCP packets
- eng8: out-of-sequence TCP packets
- eng9: bad TCP checksum packets
- eng10: outbound packets
- rtr1: inbound packets
- rtr2: client outbound packets
- rtr3: bypass outbound packets
- rtr4: dropped packets

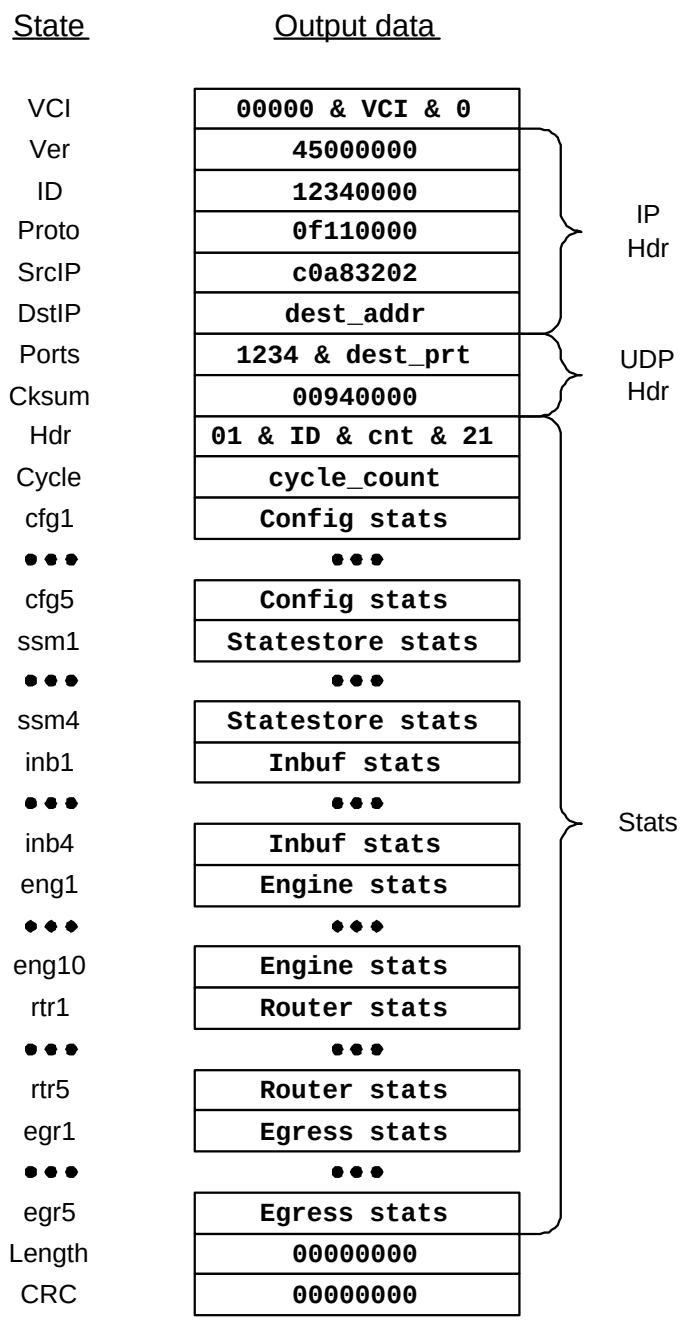


Figure 6.18: Stats Packet Format

- rtr5: TCP data bytes
- egr1: inbound client packets
- egr2: outbound client packets
- egr3: inbound bypass packets

- egr4: outbound bypass packets
- egr5: TCP checksum updates

Chapter 7

StreamExtract Circuit

The StreamExtract circuit provides primary processing of network packets in multi-board TCP flow monitoring solutions. As the name implies, it *extracts* TCP byte streams from the network traffic and passes to a separate device for additional processing. Network traffic enters the circuit through the RAD Switch interface and is processed by the Control Cell Processor (CCP) and the TCP-Processor. The StreamExtract circuit connects to the TCP processor as the client TCP flow monitoring application. This component encodes/serializes the data provided through the client interface and passes it out through the RAD Line Card interface. This traffic can then be routed to other hardware devices which perform TCP stream processing. Data returns via the RAD Line Card interface in this same format to the StreamExtract circuit where it is decoded/deserialized and returned to the TCP-Processor. After egress processing by the TCP-Processor, network traffic passes to the RAD Switch interface. Additional information regarding the StreamExtract circuit can be found at the following web site: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/stream_extract.html`

Figure 7.1 shows the layout of the StreamExtract circuit. The top level rad_streamextract entity interfaces directly with the I/O pins on the FPX platform, the VHDL code for which is located in the `rad_streamextract.vhd` source file. All external signals are brought through this interface. A one-clock-cycle buffer delay is added to all of the UTOPIA I/O signals which carry network traffic. The sram_req_fixup process ensures that the CCP module has priority access to the dual ported SRAM memory interface. This means that external access will be available to memory, even when another component locks up while controlling the memory device.

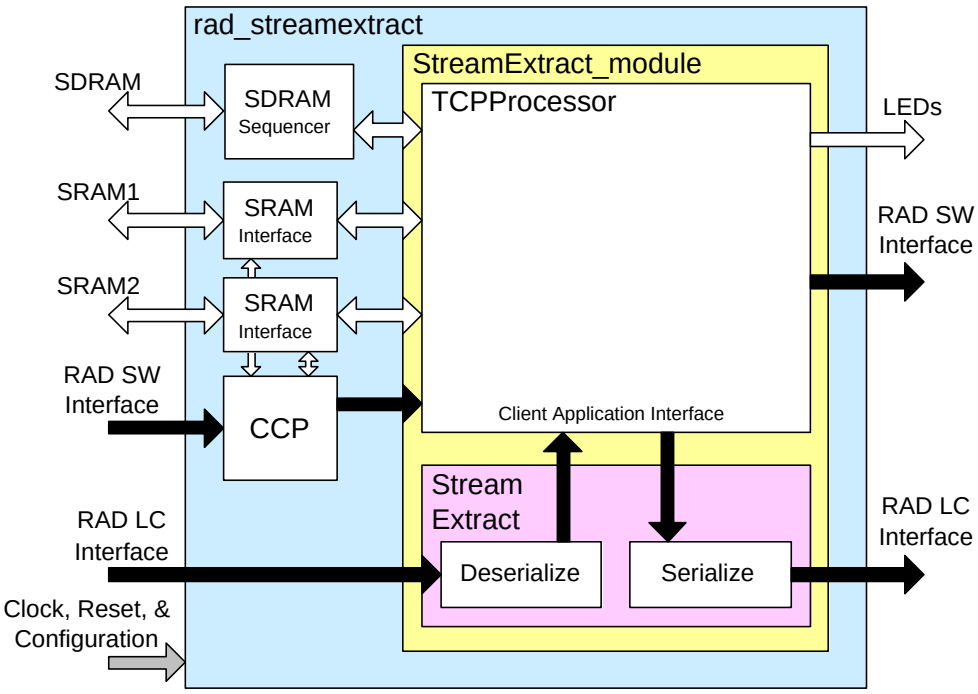


Figure 7.1: StreamExtract Circuit Layout

The remainder of the source glues the various components together. The SDRAM sequencer and SRAM interface memory controllers provide multiple interfaces and simplify access to the external memory devices. All of the I/O signals are then passed into the StreamExtract_module.

7.1 StreamExtract_Module

The StreamExtract_module component encapsulates the interaction between the TCPProcessor and the StreamExtract components. Both the TCPProcessor and the StreamExtract components contain two SRAM interfaces which can be used to capture network traffic for debugging purposes. Since there are only two SRAM devices on the chip, the StreamExtract_module contains two NOMEM dummy memory interfaces which connect to two of the four memory interfaces. These NOMEM interfaces act like memory devices which are always available, but the data is never written anywhere and read operations always return the value zero.

The `streamextract_module.vhd` source file also contains all of the configuration information for the circuit. After modifying the configuration signal values in this

file, the circuit must be rebuilt in order for the changes to be incorporated. The specific function of each configuration parameter and possible values are described in the **Configuration Parameters** section on page 76.

7.2 StreamExtract

The StreamExtract component connects the TCPSerializeEncode and TCPSerializeDecode components to the external StreamExtract interfaces. These components encode the client interface of the TCP-Processor and enable transport to other devices.

7.3 LEDs

The four RAD-controlled LEDs on the FPX platform provide a limited amount of information about the operation of the StreamExtract circuit. LED1 connects to a timer which causes the LED to blink continuously once the circuit is loaded. The rest of the LEDs indicate the current status of various flow control signals. LED2 illuminates when the flow control signal from the TCPEngine to the TCPInbuf component is low. LED3 is illuminated when the flow control signal from the TCPRouting to the TCPEngine component is low. LED4 illuminates when the flow control signal from the outbound IPWrapper to the TCPEgress module is low. During normal operation, these three LEDs are off, or at most flash only for a very brief time period. A continuously lit LED indicates that a portion of the circuit is locked-up and is not passing data.

7.4 Serialization/Deserialization (Endoding/Decoding)

The TCPSerializeEncode component processes data from the outbound client interface of the TCP-Processor. It encodes the information in a self-describing and extensible manner which supports transmitting it to other devices. Likewise, the TCPSerializeDecode component receives the encoded information and regenerates the signals required by the inbound client interface of the TCP-Processor. Figure 7.2 shows the mechanics of the encoding and decoding operations. The encoding operation compresses the packet data and extra information from the client interface signals from the TCP-Processor and groups it into several control headers which are prepended to the network packet. This information is then divided into ATM cells and transmitted through a 32-bit UTOPIA interface to other

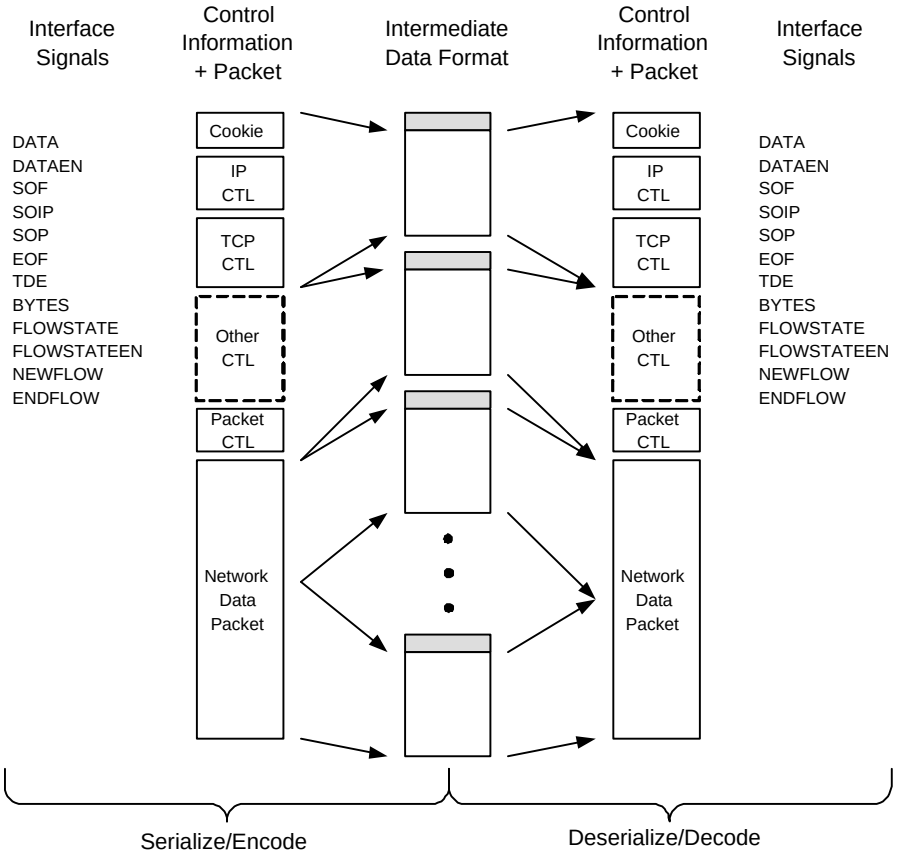


Figure 7.2: Packet Serialization and Deserialization Technique

devices. The decoding operation reverses this process, taking a sequence of ATM cells and reproducing an interface waveform identical to the one generated by the TCP-Processor.

The control header formats are described in detail utilizing standard C header file constructs. Every control header starts with an 8-bit `CTL_HEADER` which includes the type and the length of the control header, the layout of which is shown in Figure 7.3. The self-describing format of the control header makes it possible for additional control headers to be defined and added in the future without requiring a rewrite of existing circuits. This common header format allows current circuits to easily bypass control headers they don't understand. The following code sample illustrates a C style coding of the currently defined header types.

```
/* Control Header Type Definitions */

#define CTL_TYPE_UNUSED        0 #define CTL_TYPE_IP_HEADER     1
#define CTL_TYPE_TCP_HEADER    2 #define CTL_TYPE_PAYLOAD       3
```

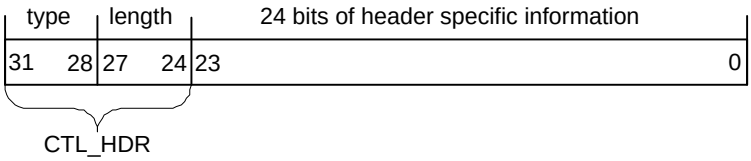


Figure 7.3: Control Header Format

```
#define CTL_TYPE_COOKIE        4 /* other headers types */

#define CTL_TYPE_RESERVED     15

struct {
    unsigned char Type   :4; /* Type code */
    unsigned char Length :4; /* Header length */
                             /*   (number of words to follow) */
} CTL_HEADER;
```

Every packet encoded by these routines is an IP packet, since the TCP-Processor receives only IP packets. All encoded packets have the same sequence of control headers after the encoding process including: a cookie control header (which acts as a record mark), an IP control header, possibly a TCP control header (depending on whether or not this is a TCP packet), and finally a packet control header. The following code sample illustrates the layout of each of these headers:

```
/* cookie control header */ #define CTL_COOKIE    0x40ABACAB


/* IP control header */ struct { /* 1st clock */
    unsigned int  HdrType    : 4;   /* standard header (Hdr.Length = 0) */
    unsigned int  HdrLength  : 4;   /* SOF is always high for the first word */
                                    /*   of the payload */
                                    /* DataEn is always high for the 2nd through */
                                    /*   the nth word of the payload */
    unsigned int  SOIPOffset : 6;   /* # of words past SOF to assert SOIP signal */
                                    /* SOIP is high for one clock cycle */
    unsigned int  SOPOffset  : 4;   /* # of words past SOIP to assert SOP signal */
                                    /* SOP remains high until the EOF signal */
    unsigned int  EOFOffset  : 14;  /* # of words past SOP to assert EOF */


} CTL_IP_HDR;


/* TCP control header */ struct { /* 1st clock */
```

```
    CTL_HEADER Hdr;                 /* standard header (Hdr.Length = 3 or 0 */
                                    /*   depending on value of ValidFlow) */
    unsigned char spare1     : 3;   /* spare */
    unsigned char ValidFlow  : 1;   /* Indicates whether or not the flow ID */
                                    /*   has been computed for this packet */
                                    /* if this value is zero, then the */
                                    /*   Hdr.Length field is set to 0 and no TCP */
                                    /* control information follows this header */
    unsigned char TDEOffset   : 4;  /* # of words past SOP to assert TDE signal */
                                    /* TDE remains high until the EOF signal */
    unsigned char StartBytes  : 4;  /* Valid bytes at start */
    unsigned char EndBytes    : 4;  /* Valid bytes at end */
    unsigned char Instance;         /* Flow instance */

/* 2nd clock */
    unsigned short StartOffset;     /* Offset from TDE to where to start */
                                    /*   supplying ValidBytes data */
    unsigned short TCPDataLength;   /* Number of TCP data bytes contained */
                                    /*   within packet */

/* 3rd clock */
    unsigned int NewFlow      : 1;  /* Indicates that this is the start of a */
                                    /*   new flow */
    unsigned int EndFlow      : 1;  /* Indicates that this is the end of an */
                                    /*   existing flow */
    unsigned int FlowID       :30;  /* Flow Identifier */

/* 4th clock */
    unsigned int NextSequence;      /* Next sequence identifier */

} CTL_TCP_HDR;


/* packet control header */ struct { /* 1st clock */
    CTL_HEADER Hdr;                 /* standard header (Hdr.Length = 0) */
    unsigned char spare;            /* spare */
    unsigned short DataLength;      /* Number of data words to follow */

/* 2nd -> nth clock */              /* payload data words */

} CTL_PAYLOAD;
```

7.4.1 TCPSerializeEncode

The `tcpserializeencode.vhd` source file contains all of the circuit logic required to encode signals from the TCP-Processor client interface into a protocol which transports easily to other devices. Figure 7.4 shows the three separate state machines that drive the

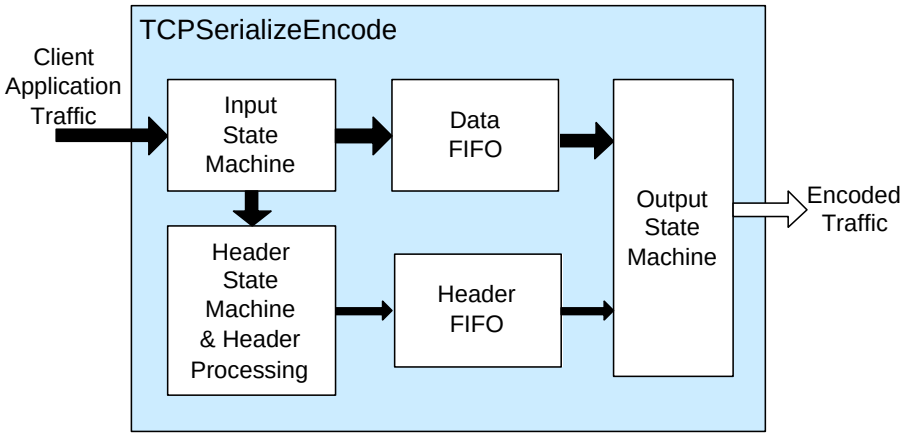


Figure 7.4: TCPSerializeEncode Circuit Layout

operation of this circuit: an input state machine, a header state machine, and an output state machine.

The first several processes in the source file consist of inactive counters. These counters were utilized during debugging and are not required for correct operation of the circuit. The first active process in the source file copies the signals from the TCP-Processor client interface to internal signals. The input state machine tracks the beginning and the end of every packet. The next process utilizes this state information to insert packet data into the data FIFO. Because the TCP-Processor contains a store and forward cycle, there are no gaps in the in the data. More specifically, all words of a packet arrive on consecutive clock cycles. For this reason, the input data processing does not need to watch the data enable signal for detecting valid data.

The next series of processes extract or compute information necessary to produce the control headers of the encoded data format. This information includes offsets from the start of the packet to the IP header, the IP payload, the end of packet, and the TCP payload data. In addition, the data delivered via the flowstate bus is stored. The header state machine drives the generation of the flowstate headers from the information collected and computed by the previous processes. The control logic which inserts the control headers into the header FIFO follows the state machine.

The output state machine generates the outbound ATM cells of the encoded format. Processing begins when the output state machine detects a non-empty header FIFO. The process generates ATM adaptation layer zero (AAL0) cells since the encoding format assumes that no cells will be lost during transmission between multiple devices. These ATM

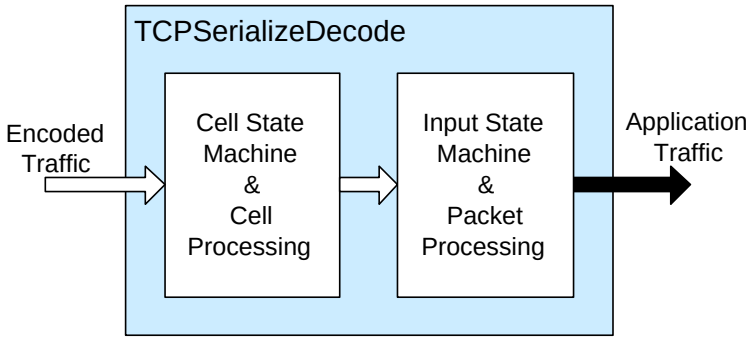


Figure 7.5: TCPSerializeDecode Circuit Layout

cells contain header information read out of the header FIFO and then packet data read out of the data FIFO.

The final set of processes contains logic for saving the encoded packet data contained in ATM cells to external SRAM. This feature aids in debugging encoding, decoding or data processing problems. All of SRAM initializes to zero prior to packet processing. The information stored in SRAM is identical to the encoded data transmitted out the RAD Line Card interface by the StreamExtract circuit.

7.4.2 TCPSerializeDecode

The `tcpserializedecode.vhd` source file contains the logic for the TCPSerializeDecode circuit. This circuit takes packets which have been encoded into ATM cells and reproduces the waveform required by the TCPEgress component of the TCP-Processor. Figure 7.5 illustrates the basic layout of this circuit.

The first process in this source file copies the inbound interface signals into internal signals used for the rest of the data processing. The cell state machine processes every ATM cell, differentiating between the cell header and the cell payload. It passes the cell payload downstream for additional processing.

The input state machine tracks the progress through the encoded control headers and the packet payload. It must detect the cookie control header as the first word of an ATM packet to initiate processing. If the cookie is detected, the state machine enters the header processing state. It transitions into the payload state when it detects the payload control header.

The next group of processes extract information out of the control headers and generate the associated packet control signals: such as the start of IP header, the start of IP

payload, the start of frame, the end of frame, and the data enable signal. They produce these output signals by counting down a packet offset count until the appropriate signal is generated.

As with the TCPSerializeEncode circuit, the TCPSerializeDecode circuit contains logic to write inbound ATM cells to SRAM memory. All of SRAM initializes to zero prior to packet processing. The information stored in SRAM is identical to the encoded data received by the StreamExtract circuit via the RAD Line Card interface.

7.5 Implementation

The StreamExtract has been implemented in FPGA logic and is currently operational. This circuit includes the TCP-Processor logic along with the previously mentioned encode and decode logic which supports data movement between multiple FPGA devices. The StreamExtract circuit has a post place and route frequency of 85.565 MHz when targeting the Xilinx Virtex XCV2000E-8 FPGA. The circuit utilizes 41% (7986/19200) of the slices and 70% (112/160) of the block RAMs. The device is capable of processing 2.9 million 64-byte packets per second and has a maximum data throughput of 2.7Gbps.

Figure 7.6 shows a highly constrained circuit layout of the StreamExtract circuit on a Xilinx Virtex 2000E FPGA. The layout consists of regions roughly corresponding to the various components of the circuit. The control processor contains logic for processing ATM control cells which facilitate the reading and writing of external memory devices connect to the FPGA. This component is used for debugging purposes only and is not a required part of the StreamExtract circuit. Presenting the physical circuit layout in this manner provides a good comparison of the amount of FPGA resources each component requires.

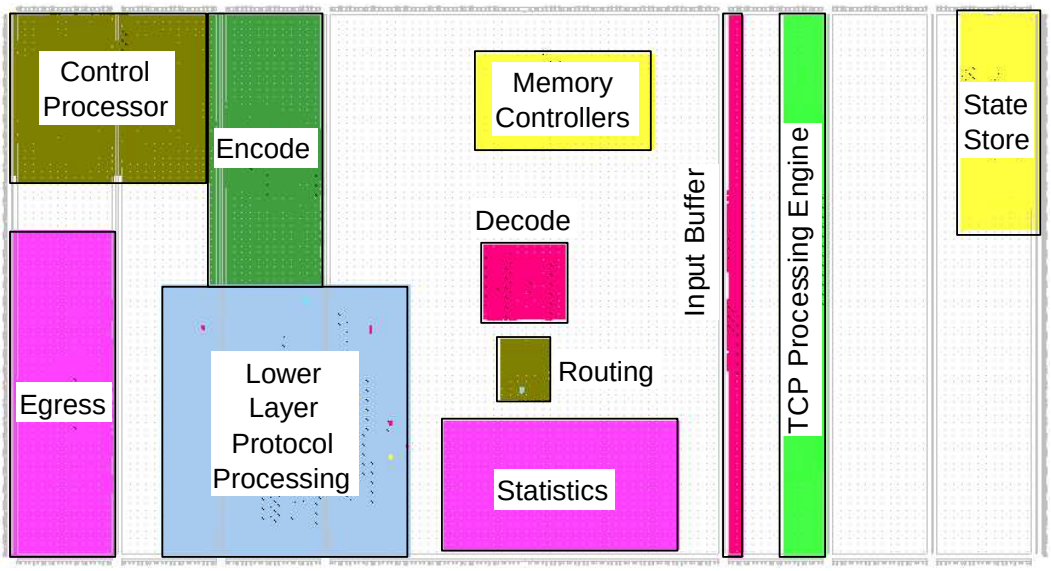


Figure 7.6: StreamExtract Circuit Layout on Xilinx XCV2000E

Chapter 8

TCP-Lite Wrappers

TCPLiteWrappers[1], composed of the TCPDeserialize and TCPReserialize circuits, provide an environment in which TCP flow processing applications can operate. The TCPDeserialize circuit converts the encoded data stream generated by the TCPSerializeEncode circuit into the original client interface waveform produced by the TCP-Processor. The TCPReserialize circuit re-encodes the output of the client TCP processing circuit for transmission to other devices. The output format of the TCPDeserialize circuit is identical to the input format of the TCPReserialize circuit. Likewise, the input format of the TCPDeserialize circuit is identical to the output format of the TCPReserialize circuit. Given the commonality of these interfaces, the pair of the TCPDeserialize and TCPReserialize circuits have a net zero effect on data that they process and could be inserted multiple times into a given application.

The source for the TCPDeserialize and TCPReserialize circuits can be found in both the distributions for *PortTracker* and *Scan*. The source files are identical in both distributions.

8.1 TCPDeserialize

The TCPDeserialize circuit is different from the TCPSerializeDecode circuit because the TCPSerializeDecode circuit only needs to reproduce a subset of the client interface signals as required by the inbound TCPEgress component. The TCPDeserialize component needs to faithfully reproduce all of the original TCP-Processor client interface signals. For this reason, the TCPDeserialize circuit contains additional logic functions when compared to

[1]The TCPLiteWrappers moniker was created by James Moscola. Previously, TCPLiteWrappers were unnamed.

the TCPSerializeDecode circuit. Aside from the few extra signals and the processses that drive them, these two circuit components are identical. The `tcpdeserialize.vhd` source file contains the source code for the TCPDeserialize circuit.

8.2 TCPReserialize

Although similar, the TCPReserialize circuit is different from the TCPSerializeEncode circuit because the TCPSerializeEncode circuit assumes that all packet data will be received in consecutive clock cycles. This condition does not hold true for the TCPReserialize circuit which contains extra logic to deal with gaps in packet transmission.

The `tcpreserialize.vhd` source file contains the source code for the TCPReserialize circuit. As stated previously, the layout, behavior, processes and signals for this circuit are almost identical to those of the TCPSerializeEncode circuit. Please refer to the documentation on the TCPSerializeEncode circuit in order to understand the operation of the TCPReserialize circuit.

8.3 PortTracker

The PortTracker application processes TCP packets and maintains counts of the number of packets sent to or received from selected TCP ports. This circuit clarifies the distribution of packets among the various TCP ports. This circuit also provides an excellent example of how to utilize the TCPLiteWrappers when the client TCP stream processing application is not concerned about maintaining state information on a per-flow basis. The source for the PortTracker application can be found at the following URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp /source/porttracker_source.zip`. The directory structure layout, circuit layout, and build procedures for this application are similar to that of the StreamExtract circuit.

Figure 8.1 shows a layout of the PortTracker application components, including the main data flow and the interactions with SRAM devices that capture traffic to aid in the debugging process. The *rad_porttracker* component is the top level design file and interfaces directly to the I/O pins of the RAD. The VHDL code for this component can be found in the `rad_porttracker.vhd` source file. The component passes control cells and normal network packets into the circuit via the RAD Switch interface. This network traffic routes first to the Control Cell Processor (CCP) which allows remote access to the SRAM

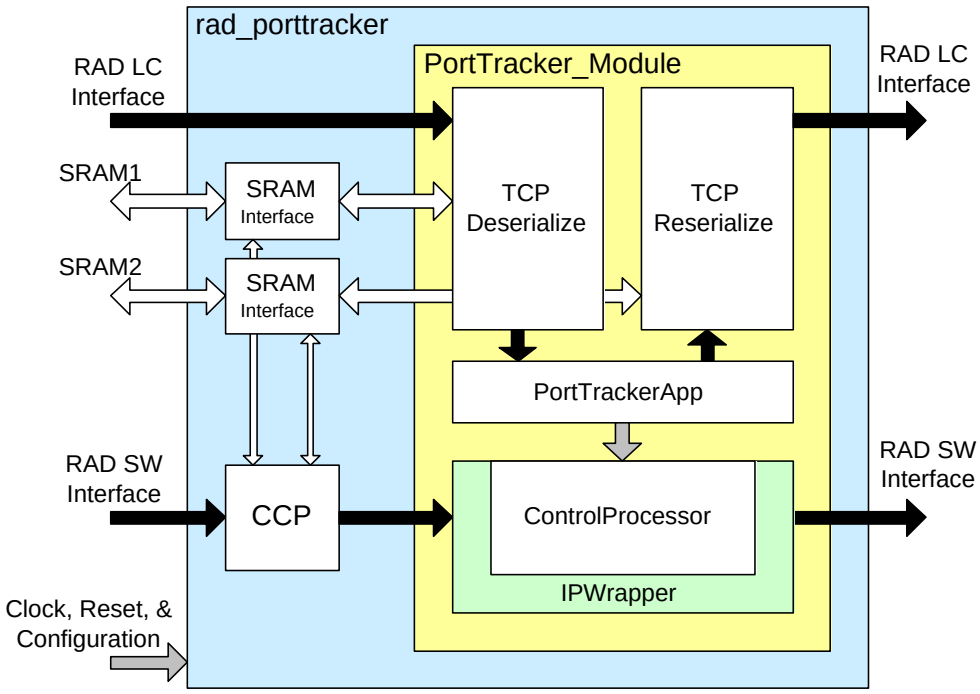


Figure 8.1: PortTracker Circuit Layout

devices. Encoded TCP stream data enters the circuit via the RAD Line Card interface. Network data from both of the external interfaces is passed into the PortTracker_Module for processing. The SRAM Interface components expose multiple memory access ports, allowing for request multiplexing. This makes it possible for multiple components to share the same memory device.

8.3.1 PortTracker_Module

The VHDL code for the PortTracker_Module can be found in the source file, `port-tracker_module.vhd`. This component acts as the glue which holds all the subcomponents of the PortTracker circuit together and provides data routing and signal connections between the subcomponents and the external interface. The RAD Switch interface passes data to the IPWrapper which converts the inbound cells into IP packets which are then passed to the ControlProcessor. The ControlProcessor passes outbound traffic back to the IPWrapper which converts packets back into ATM cells for transmission out of the RAD Switch interface. Encapsulated TCP stream data enters through the RAD Line Card interface and goes to the TCPDeserialize circuit. The TCPDeserialize circuit passes annotated

TCP packets to the PortTrackerApp circuit which scans network traffic for packets sent to or received from various TCP ports. Network traffic is then routed to the TCPReserialize circuit which re-encodes the network packets for transmission out of the RAD Line Card interface.

The PortTracker_Module contains the same configuration parameters that define the operation of the PortTracker application. These configuration parameters are identical to those described in the StreamExtract circuit. Refer to the previous **Configuration Parameters** section on page 76 for details on each of the configuration parameters.

8.3.2 PortTrackerApp

The implementation for the PortTrackerApp can be found in the `porttracker.vhd` source file. The PortTracker application is a very simple circuit and a good example of how to implement a basic TCP packet processing application. The first process copies the inbound data into internal signals used in subsequent processes. The *istcp* signal indicates whether or not the current packet is a TCP packet. The *hdr_count* signal is a remnant from previous logic and is not used in the application. The *port_flop* process performs all of the work for the PortTracker application. The *SOP* and *istcp* signals indicate the presence of the TCP port number on the data bus. When this situation occurs, the source and destination TCP ports run through a series of *if* statements which determine the type of packet being processed. The result of the port comparisons, drives a port type signal. Packets are always associated with the lowest port match (either source or destination) and are only counted once. The PortTrackerApp passes the various port type signals to the ControlProcessor for collection and generation of statistics packets.

The last group of processes in the PortTrackerApp copies the internal packet data signals to the external interface. Additionally, a running counter and a process uses this counter to flash the RAD LEDs. These LEDs indicate that the circuit has been loaded into the FPGA on the FPX platform.

8.3.3 ControlProcessor

The ControlProcessor component is described in the `control_proc.vhd` source file. Figure 8.2 shows the flow of data through the circuit. The first several processes copy inbound network data into internal signals and load the packets into the Frame FIFO. Once a complete packet has been written to the Frame FIFO, one bit is written to the control FIFO to indicate that a complete packet is ready for outbound transmission.

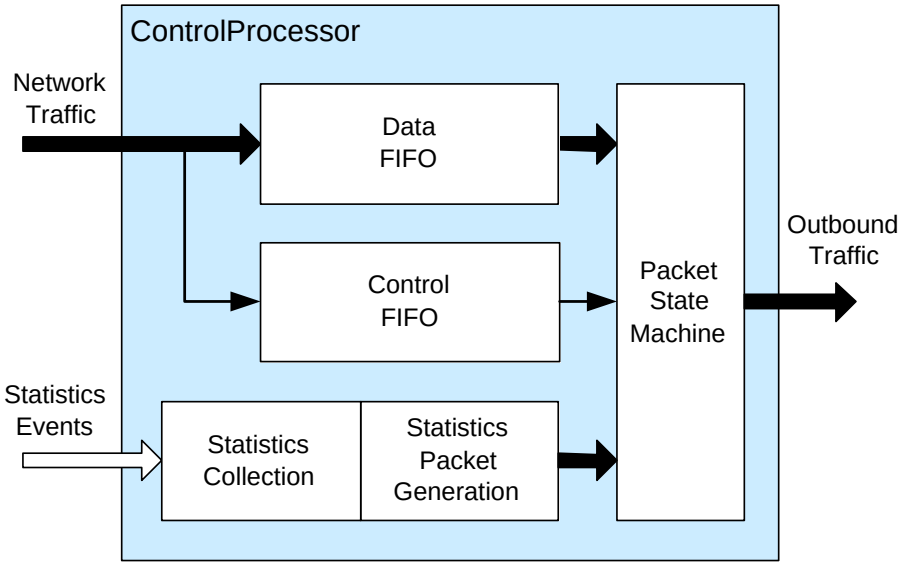


Figure 8.2: PortTracker ControlProcessor Layout

The next couple of processes maintain the interval timer which drives the generation of the trigger event. This operation is very similar to that of the TCPStats module of the TCP-Processor. The trigger event initiates the saving of all statistics counters and the transmission of a statistics packet.

The next group of processes maintain statistics counters for each of the port ranges tracked by this circuit. These counters are saved and reset when they receive the trigger event.

The packet state machine manages the transmission of outbound traffic. Upon detection of the trigger event, the state machine traverses a sequence of states that generate the statistics packet and send it out via the outbound interface. When it detects a non-empty condition for the control FIFO, it reads a network packet from the frame FIFO and transmits it out of the output interface.

The next process takes a snapshot of all the statistic counters and saves the current values when the trigger event fires. This allows the counters to reset and continue counting the generation of the statistics packet.

The format of the statistics packet follows the same general format as mentioned in Appendix C on Page 183. The PortTracker application generates a statistics packet with a stats identifier of five (5). The exact sequence of statistics values transmitted in the statistics packets is:

- cfg1: simulation
- numFTP: number of FTP packets
- numSSH: number of SSH packets
- numTelnet: number of telnet packets
- numSMTP: number of SMTP packets
- numTIM: number of TIM packets
- numNameserv: number of Nameserv packets
- numWhois: number of Whois packets
- numLogin: number of Login packets
- numDNS: number of DNS packets
- numTFTP: number of TFTP packets
- numGopher: number of Gopher packets
- numFinger: number of Finger packets
- numHTTP: number of HTTP packets
- numPOP: number of POP packets
- numSFTP: number of SFTP packets
- numSQL: number of SQL packets
- numNNTP: number of NNTP packets
- numNetBIOS: number of NetBIOS packets
- numSNMP: number of SNMP packets
- numBGP: number of BGP packets
- numGACP: number of GACP packets
- numIRC: number of IRC packets
- numDLS: number of DLS packets
- numLDAP: number of LDAP packets
- numHTTPS: number of HTTPS packets
- numDHCP: number of DHCP packets
- numLower: number of packets to/from TCP ports ≥ 1024
- numUpper: number of packets to/from TCP ports > 1024

8.4 Scan

The Scan application processes TCP data streams and searches for four separate digital signatures of up to 32 bytes in length. This circuit provides TCP stream content scanning features which support detection of signatures that cross packet boundaries. This circuit also

provides an excellent example/model/starting point of how to utilize the TCPLiteWrappers when the client TCP stream processing application needs to maintain context information for each flow in the network. The source for the Scan application can be found at the following URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/source/scan_source.zip`. The directory structure layout, circuit layout, and build procedures for this application are similar to that of the StreamExtract and PortTracker circuits. Additional information regarding the Scan circuit can be found at the following web site: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/scan.html`

Figure 8.3 shows a layout of the components which make up the Scan application. It illustrates the main data flow along with interactions it has with memory devices which capture traffic to aid in the debugging process and store per-flow context information. The *rad_scan* component is the top level design file which interfaces directly to the I/O pins of the RAD. The VHDL code for this component can be found in the `rad_scan.vhd` source file. The RAD Switch interface passes control cells and control packets into the circuit. This data goes first to the Control Cell Processor (CCP) which allows remote access to the SRAM devices. Encoded TCP stream data enters the circuit via the RAD Line Card interface. Network data from both of the external interfaces is passed into the Scan_Module for processing. The SRAM Interface component exposes multiple memory access ports to provide request multiplexing so that multiple components can share the same memory device.

8.4.1 Scan_Module

The VHDL code for the Scan_Module is in the `scan_module.vhd` source file. This component acts as the glue which holds all the subcomponents of the Scan circuit together and provides data routing and signal connections between the subcomponents and the external interface. The RAD Switch interface passes data to the IPWrapper which converts the inbound cells into IP packets which are then passed to the ControlProcessor. The ControlProcessor passes outbound traffic back to the IPWrapper which converts packets back into ATM cells for transmission out of the RAD Switch interface. Encapsulated TCP stream data enters through the RAD Line Card interface which passes it to the TCPDeserialize circuit. It passes annotated TCP packets to the ScanApp circuit which scans network traffic for four separate digital signatures. Network traffic is then routed to the TCPReserialize

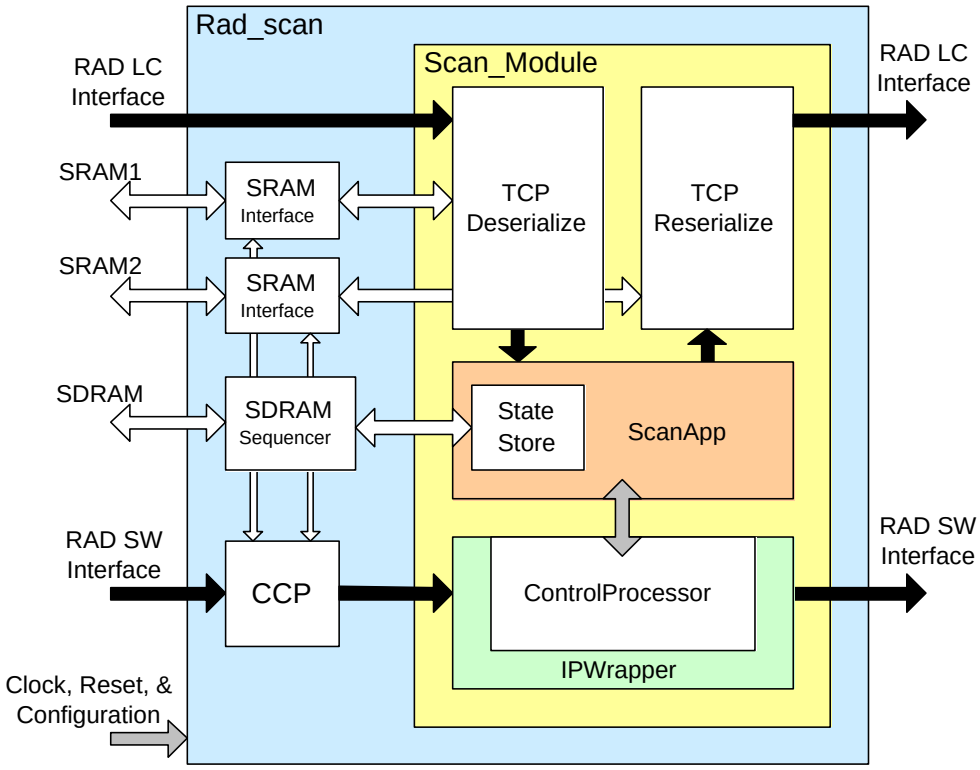


Figure 8.3: Scan Circuit Layout

circuit which re-encodes the network packets for transmission out of the RAD Line Card interface.

The Scan_Module also contains the configuration parameters which define the operation of the Scan application. These configuration parameters have identical names and functions to those described in the StreamExtract and PortTracker circuits. Please refer to the previous Configuration Parameters section on page 76 for details on each of the configuration parameters.

8.4.2 ScanApp

The implementation for the ScanApp can be found in the `scan.vhd` source file. Figure 8.4 shows the internal logic layout of the ScanApp. The main flow of network traffic goes left to right. The output state machine controls the operation of the stream comparison logic. The StateStore component stores the most recent 32 bytes of stream data for each flow in external memory. It loads this per-flow context information into the string comparison logic

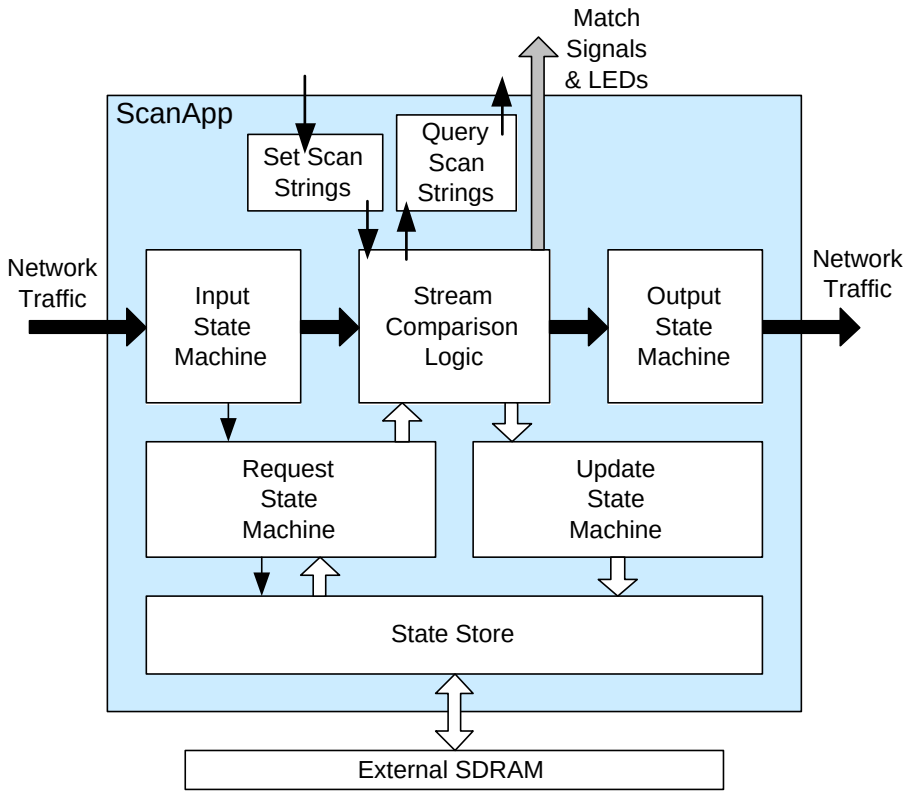


Figure 8.4: ScanApp Circuit Layout

prior to processing a packet. After packet processing, it saves this context information to memory. It also possesses logic to control the replacement of search strings.

The first couple of processes of the ScanApp perform component maintenance tasks. These tasks include controlling the TCA flow control signal for inbound data. The TCA signal de-assertes whenever the data FIFO becomes 3/4 full or the control or header FIFOs become 1/2 full. They also copy inbound data into internal signals used later in the circuit.

The input state machine manages the processing of inbound packets into the Scan circuit. Based on the input processing state, it writes packet data into the data FIFO and flowstate information and control signals into the header FIFO. When it detects the end of a packet, it writes a single bit to the control FIFO which indicates that a complete packet is available for downstream processing.

The next group of processes handles the processing of header information, issuing requests to the StateStore to retrieve per-flow context information, and processing the returned context information. The request state machine drives the sequence of operations required to interact with the StateStore. Upon detecting a non-empty header FIFO, the state

machine progresses through states which retrieve flowstate information from the header FIFO. Depending on information it retrieves from the header FIFO and the packet's relation to the previous packet (is this packet part of the same flow or not), the state machine will enter the WFR (wait for response) state, waiting for completion of the per-flow context retrieval from the StateStore. It enters the WFO (wait for outbound processing) state last, which ensures that data is held in internal buffers until outbound processing is initiated. The hfifo_control process controls the reading of information out of the header FIFO. The req_signals process stores up to four words of flowstate information read from the header FIFO. The state_store_request process drives the StateStore request signals which initiate the reading of per-flow context information. The retrieval_active flop indicates whether or not a retrieval operation of per-flow context information is active for this packet. Context information returned from the StateStore is saved in a set of signal vectors, prior to being loaded into the comparator.

The output state machine drives the processing of the stream comparison engine and the delivery of network traffic out of the component. When it detects a non-empty condition for the control FIFO and the request state machine is in the WFO state (indicating that StateStore request processing has completed for the next packet), the output state machine initiates outbound packet processing. It enters a delay state in order to wait for data to be clocked out of the data FIFO. The first four words of a packet have special states (First, Second, Third and Fourth). This supports the special processing required to combine the flowstate information with the network packet. The Payload state manages the processing of the remainder of the packet. The Last1 and Last2 states handle processing of the AAL5 trailer which occurs at the end of the network packet. A couple of additional processes control the reading of data from the data and control FIFOs. The next process drives internal signals with packet data, flowstate data, and associated control signals.

The next three processes store information required by the StateStore update logic after completion of packet processing. They use separate signals to store this information so that the front end processing (input and request state machines) can initiate processing of the next packet. These signals store the flow identifier associated with this packet, a determination of whether or not an update is required, and an indication of whether or not this is a new flow. If either the update required or the new flow signals are set, the update state machine will store the current flow context to memory via the StateStore component.

The next group of processes maintains the check string which contains the most recent 36 bytes of the TCP data stream for this flow. The valid_byte_count signal keeps track of how many new bytes of data (0, 1, 2, 3, or 4) are added to the check string.

The build_check_string process maintains an accurate representation of the most recent 36 bytes of the TCP data stream. While processing the first word of the packet, the build_check_string process loads the check string with data either retrieved from the StateStore or initialized to zero. Subsequently, whenever there is valid TCP stream data on the data bus, the check string is updated accordingly. Since data is clocked in four bytes at a time, the process maintains a 36-byte check buffer so that it can perform the four 32-byte comparisons on each clock cycle, shifting each comparison by one byte.

The next section of logic performs update operations to the StateStore. The update state machine controls this action. The update sequence starts when the update state machine reaches the end of a packet and either the upd_required signal or the upd_newflow signal indicate that context information should be saved. Once begun, the sequence continues to cycle to a new state on each clock cycle until it is complete. This sequence involves sending the flow identifier, followed by four 64-bit words of per-flow context information. The ss_out signals temporarily store the last 32 bytes of the check string. This allows the check string to be loaded with the context information of the next packet without having to wait for the context update operation to complete. The state_store_update process drives the StateStore signals utilized to save the new per-flow context information.

The next section of logic contains the stream comparison logic. A TCAM approach performs the actual comparisons where each comparator contains a value and a mask. It performs the match, checking to see if the following equation is true:

$$((check\ string\ XOR\ value)\ NAND\ mask)\ ==\ (all\ ones)$$

The first process controls all of the matching operations by indicating whether a match should be performed on this clock cycle. The next process maintains the match1_value and match1_mask values. These values update when the set_state and set_id signals are set. The 256-bit string 1 match operation is divided into eight segments in order to reduce the fan-in for the match_found signal. Each of these eight processes match a separate 32 bits of the match1 string and produce a separate match_found1 signal. The match process performs four separate matches, each offset by one byte in order to achieve full coverage for the matching operation. An identical set of processes manages the matching operations for strings 2, 3, and 4.

The next processes drive the outbound signals by copying the internal packet signals. This induces an extra clock cycle delay in the data, but helps to improve the maximum

operating frequency by providing an extra clock cycle for these signals to traverse the FPGA.

The next group of processes illuminates the LEDs whenever one of the match strings is detected in a TCP data stream. The first process maintains a free-running counter which is used by the subsequent processes to define the duration that a LED should be lit on the occurrence of a string match. This is required for the LED to be visible (illuminating the LED for 1 clock cycle has no visible effect). The led1_count and the led1_ena signals manage the illumination of LED 1. When all of the match1_found signals are set, the led1_count signal stores the current counter value and illuminates the LED by setting the led1_ena signal. The LED remains illuminated until the counter wraps and returns to the value saved in the led1_count signal. LEDs 2, 3, and 4 operate the same way.

The next series of processes manage the modification of the search strings. The set state machine controls the sequence of events which waits for the SET_ENA signal to be asserted. The first word of SET_DATA contains the number of the string to be modified. The next eight clock cycles contain the string value and the final eight clock cycles contain the string mask. The set_count signal counts through the eight clock cycles associated with the string value and mask. The set_value signal holds the new match value and the set_mask holds the new match mask. If it receives an incomplete set sequence, the set state machine returns to the idle state and ignores the command.

The query state machine manages the querying of the current match string values and masks. The ControlProcessor component driving the QUERY_ENA signal and providing the identifier of the string to query on the QUERY_DATA signals initiates the query request. On receiving the QUERY_ENA signal, the ControlProcessor saves the current match string value and mask values into the qry_value and qry_mask signals. During the next eight clock cycles, The ControlProcessor clocks the match value over the RESPONSE_DATA signals. It then clocks the match mask over the same response signal lines during the subsequent eight clock cycles. The ControlProcessor uses the RESPONSE_ENA and RESPONSE_LAST signals as extra control information to frame the query response.

The final process of the source file manages miscellaneous output signals. These include the output LEDs and a set of match signals which are used by the ControlProcessor to collect string match statistics.

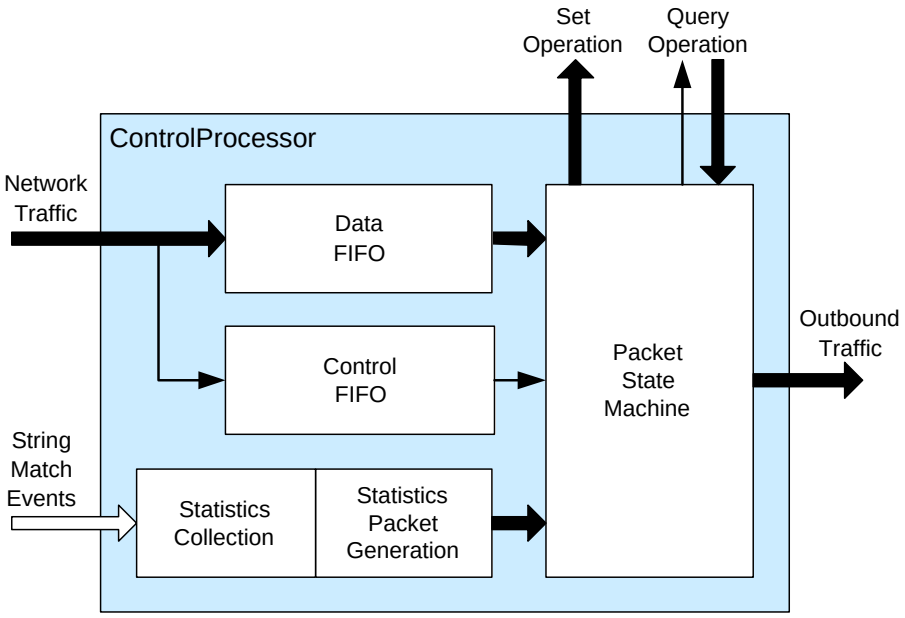


Figure 8.5: Scan ControlProcessor Layout

8.4.3 ControlProcessor

The `control_proc.vhd` source file describes the ControlProcessor component and the overall flow of data through the circuit. The first several processes copy inbound network data into internal signals and load the packets into the frame FIFO. Once a complete packet has been written to the frame FIFO, one bit is written to the control FIFO to indicate that a complete packet is ready for outbound transmission.

The next couple of processes maintain the interval timer which drives the generation of the trigger event. This operation is identical to that of the ControlProcessor module of the PortTracker application. The trigger event initiates the saving of all statistics counters and the transmission of a statistics packet. The next group of processes maintain statistics counters for each of the four string matches which are tracked by this circuit. On reception of the trigger event, the ControlProcessor saves and resets these counters.

The packet state machine processes match string set commands, match string query requests, and manages the transmission of all outbound traffic (including the forwarding of inbound traffic, responses to commands and queries, and generated statistics packets). Upon detection of the trigger event, the state machine traverses through a sequence of states which generate the statistics packet and send it out via the outbound interface. When the state machine detects a non-empty condition for the control FIFO, it reads and processes a

network packet from the frame FIFO. If the packet is determined to contain a command or query for the Scan application, then the state machine traverses a sequence of states which either process the match string set command or the match string query command. The state machine makes this determination by looking at the source and destination ports of the incoming packet. If both of these port numbers are equal to `0x1969`, then the packet state machine will enter the command/query processing states. The state machine also looks at the first word of the UDP payload to see if it contains the *magic* value of `0x23a8d01e`. If this *magic* value is detected, then the next word is assumed to be the command word which determines whether this command is a set or a query. Results of the set or query operation generate an outbound packet. The state machine transmits all other traffic out of the output interface. An additional couple of processes control the reading of packets from the frame and control FIFOs. The last_read signal marks the end of a packet and is used by the frame FIFO output control process along with the ffifo_ack signal to ensure that all packet data is read from the frame FIFO.

The next process takes a snapshot of all the statistic counters and saves the current values when the trigger event fires. This allows the counters to reset and continue counting while the statistics packet is being generated and transmitted.

The format of the statistics packet follows the same general format as mentioned in Appendix C on Page 183. The Scan application generates statistics packet with a stats identifier of two (2). The exact sequence of statistics values transmitted in the statistics packets is listed below:

- cfg1: simulation
- scn1: number of occurrences of match string 1
- scn2: number of occurrences of match string 2
- scn3: number of occurrences of match string 3
- scn4: number of occurrences of match string 4

The output process drives the outbound signals with network packets. There are four basic packet types. The first is statistics packets, the second is a response to the set command, the third is a response to the query command, and the fourth is passthrough traffic. The ControlProcessor statistic packet generation logic is identical to the statistic packet generation logic contained in the PortTracker and TCP-Processor circuits, except for the individual statistics values. In order to simplify some of the command processing logic, all inbound packets to the ControlProcessor are assumed to be control packets. For all of these packets, the output process replaces the source IP address with the destination

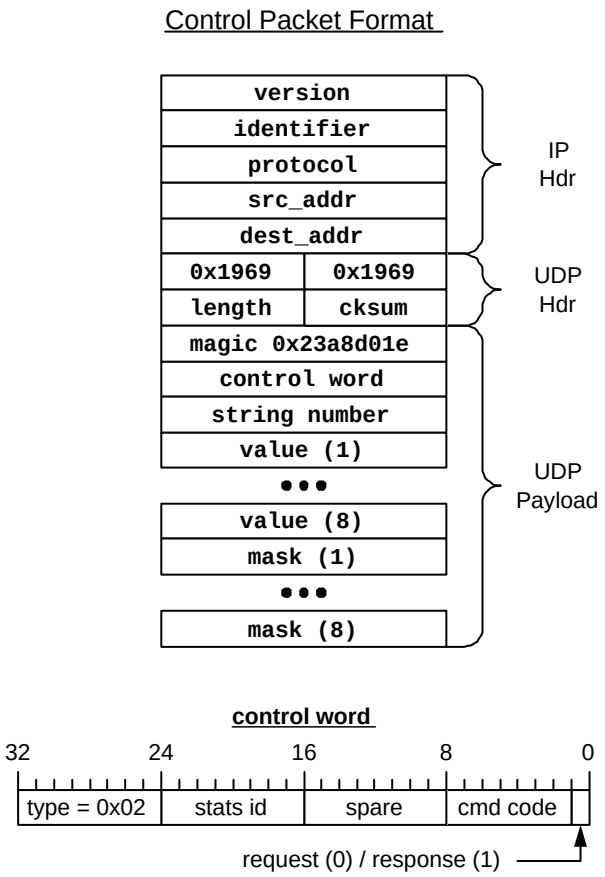


Figure 8.6: Scan Control Packet Format

IP address and inserts the specified stats source IP address into the source IP address field of the packet. Additionally, the packet length of the UDP header is changed to 0x54 to correspond to the length of a response/confirmation packet. During the command state, the request/response bit is set in the control word. The result of the query operation is output during the WFR and the Response states. The layout of the Scan control packet format can be seen in Figure 8.6. All other inbound packet data is forwarded to the output interface in a normal fashion.

Following the output process, there are a couple of processes which aid in the processing of control packets. These processes count the number of IP header words, store the destination IP address so that it can be used to replace the source IP address, and generate statistics packets. The final two processes drive the SET and QUERY interface signals to the ScanApp component. These signals actually control the set and query operations.

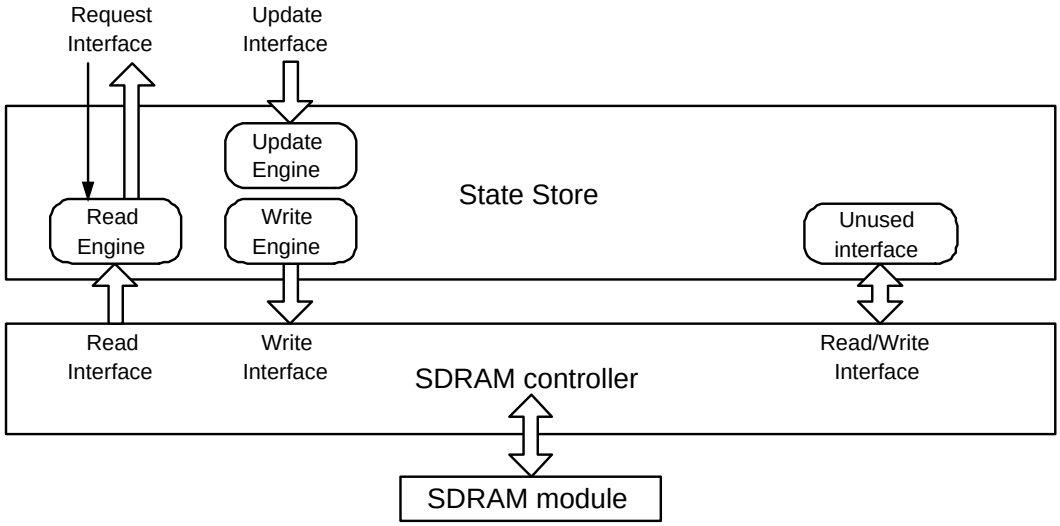


Figure 8.7: Layout of StateStore

8.4.4 StateStore

The StateStore component provides per-flow context storage services for the Scan application. It exposes simple request, response and update interfaces to the application. In addition, this component handles all interactions with the SDRAM controller to store and retrieve data. The StateStore is modelled after the StateStoreMgr of the TCP-Processor, but is greatly simplified because it does not perform any of the flow classification operations. It only provides data storage and retrieval services for the Scan application. In order to simplify the operation of the StateStore component, 64-bit wide interfaces are utilized for both the response and update data.

Figure 8.7 shows the layout of the StateStore component. Its three state machines drive all of the operations of the StateStore. The source VHDL code for the StateStore component is in `statestore.vhd`.

The first process in the source file drives the response interface signals. It copies data returned from SDRAM to the response interface signals and passes it to the ScanApp component. The next couple of processes implement the read state machine. Upon receiving a request from the ScanApp, the state machine sequences through the states necessary to perform the memory read operation and receive the response. The read engine drives the read control signals of the SDRAM controller. The SS_REQ_DATA signal contains the flow identifier of the context to retrieve. This flow identifier is used as a direct address into memory and the read engine initiates the 32-byte read operation starting at this address.

This flow identifier (memory address) is stored in the read_addr signal for the duration of the read operation. Once the presence of the read response has been signaled by the memory controller, the take_count signal counts the data words returned from memory. As this data is being returned from memory, it is copied to the response signals and passed to the ScanApp component.

The update state machine processes data associated with an update request. This component requires a separate state machine and buffer because write data cannot be written directly to the SDRAM controller without first setting up the write operation. Upon detection of an update command, the new context information is written to the writeback_ram for buffering while the write operation is being initiated. The update_count signal tracks the number of received words from the ScanApp and is also used as the address of where to write the data in the writeback_ram block.

The write state machine manages write operations to the SDRAM controller. The initial sequence of states out of reset perform memory initialization tasks by writing zeros to all memory locations. After memory initialization is completed, the write state machine enters the Idle state, waiting for update requests. On receipt of the SS_UPD_VALID signal, the state machine enters the Req state to initiate the write request to SDRAM. Based on responses from the SDRAM controller, the write state machine traverses through the remaining states of the write operation. The next process controls the reading of the updated per-flow context information from the writeback_ram. It uses a separate interface so that writes to the writeback_ram can occur at the same time as read operations. The write_engine drives the SDRAM controller interface signals. Operations include the initialization (zeroing) of all memory locations and the writing of context information to external memory. The write_addr signal holds the address of the appropriate context information storage location in memory for the duration of the write operation. Following the write_addr signal, the processes that control the memory initialization operation provide an incrementing memory address and a completion flag. The next process counts the number of words read from the writeback_ram and written to external SDRAM. This count value is used as the read address for the writeback_ram. The final process in the source file drives the ready signal. It indicates when the memory initialization operation completes.

Chapter 9

Analysis

The TCP processing circuits analyzed live Internet traffic for this phase of the research. Network Technology Services, which manages the campus network at Washington University, provided a live copy of all campus Internet traffic to the Applied Research Laboratory for this analysis. The Washington University network has approximately 19,000 active IP addresses and uses two separate Internet connections which provide 300Mb/s of total bandwidth. In addition, the University maintains an 84Mb/s connection to Internet2, a high-speed network interconnecting 206 universities throughout the United States. Traffic flowing over these three Internet connections was aggregated and passed to the FPGA-based TCP flow processing circuits via a Gigabit Ethernet link.

9.1 Test Setup

The test environment used for Internet traffic processing and data collection includes a WUGS-20 switch and a Linux PC. The WUGS-20 switch is populated with four FPX cards which perform data processing, two Gigabit Ethernet cards (one for injecting Internet traffic and the other for exporting statistics), and a G-Link card which is used for switch configuration operations. Figure 9.1 is a diagram of the test setup. Figure 9.1A is a photograph of the hardware used to process Internet traffic and Figure 9.1B illustrates the flow of data through the hardware.

Live Internet traffic enters the WUGS-20 switch via a Gigabit Ethernet connection. Both inbound and outbound traffic arrive on a single network feed. The Gigabit Ethernet network card passes data packets StreamExtract circuit for processing by the TCP-Processor. The StreamExtract circuit passes annotated TCP packets to the PortTracker

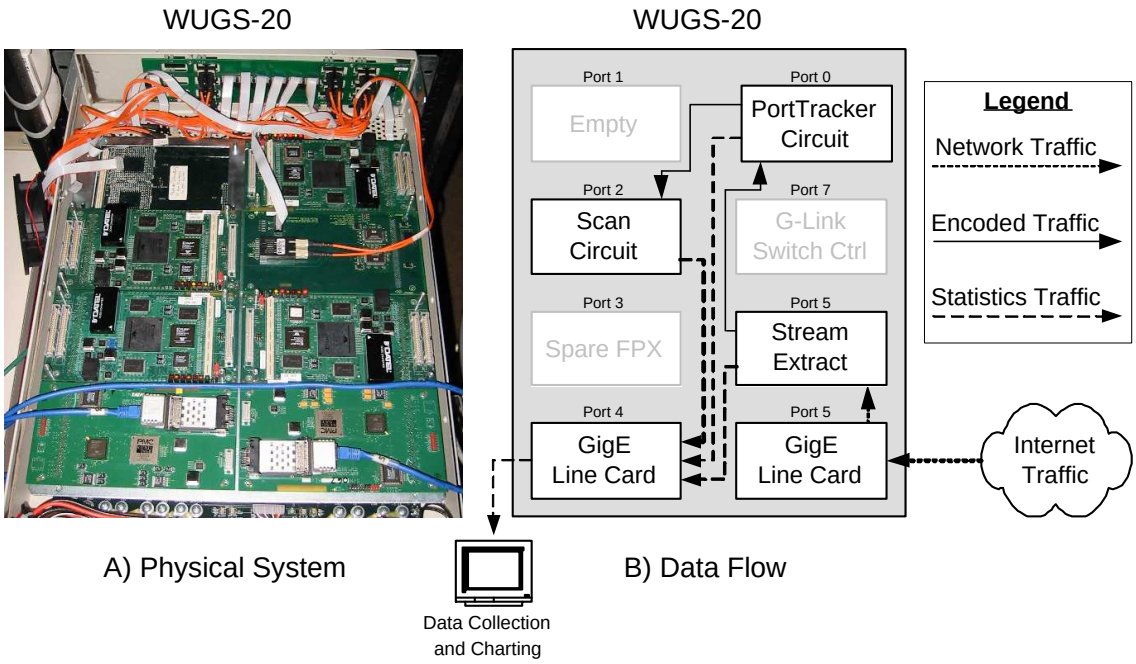


Figure 9.1: Internet Traffic Monitor Configuration

circuit which collects statistics on the amount of TCP traffic used by various well-known TCP ports. The PortTracker circuit then passes the encoded data is then passed to the Scan circuit which scans TCP data streams for selected digital signatures. Each circuit transmits statistics information to an external computer which performs data collection and charting.

9.2 Data Collection

A Linux PC connected to the WUGS-20 switch gathers statistics packets generated by the StreamExtract, PortTracker and Scan circuits. Software applications running on this PC collect, store, process, and chart the information contained in the statistics packets. The StatsCollector (Appendix C, Page 184) application receives these packets first. The purpose of the StatsCollector is to accurately time stamp each packet and store the entire contents of the packet to a disk file. The StatsCollector maintains a separate disk file for the statistics generated by each circuit. In addition, it creates a new statistics data file each day to facilitate the maintenance and processing of the raw data. The data contained in these daily files can easily be plotted using the gnuplot graphing utility. The StatsCollector

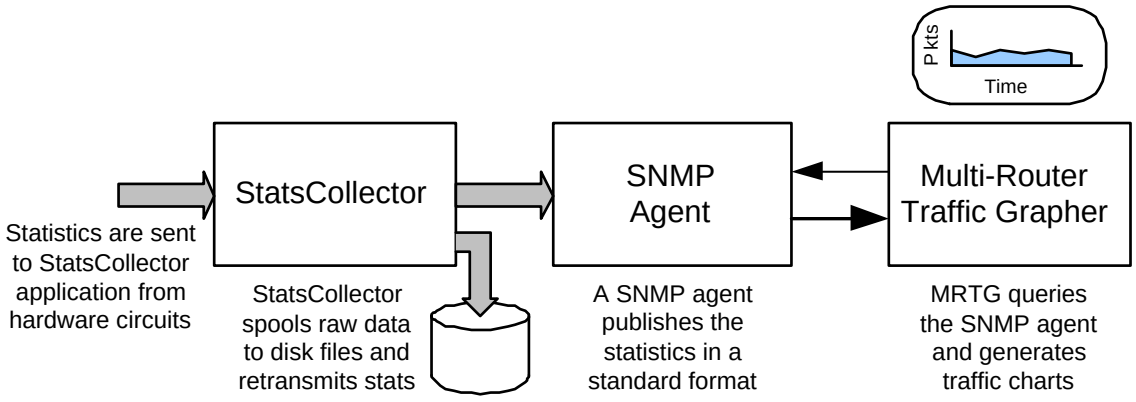


Figure 9.2: Statistics Collection and Presentation

circuit also retransmits the original packet so that other statistics processing applications can operate on data in real-time.

A Simple Network Management Protocol (SNMP) agent (Appendix C, Page 186) specifically designed for this project runs on the data collection PC. This agent receives the retransmitted statistics packets and maintains individual counters for each item in the packets. These counters are made available as Management Information Base (MIB) variables via standard SNMP queries. The Multi-Router Traffic Grapher (MRTG) is a utility which produces real-time updating charts by reading SNMP MIB variables on a periodic basis. A configuration file defines the operation of the MRTG application, specifying which MIB variables to plot and the format, legends and titles of each chart (Appendix C, Page 187). Figure 9.2 shows a diagram of the statistics data collection and processing.

9.3 Results

The testing analyzed Internet traffic for a five week period from August 20th, 2004 and to September 25th, 2004. Figure 9.3 shows IP traffic transiting the Internet drains during a 24 hour period. This daily traffic pattern is representative of the observed traffic during the entire collection period. A cyclical traffic pattern exists with peak traffic rates occurring between 12:00 noon and 5:00 pm and a trough occurring between 3:00 am and 8:00 am. The testing detected peak traffic rates of 300Mbps. While this traffic load does not tax the throughput capabilities of the circuit, it does demonstrate data processing on live Internet traffic at rates above what software-based TCP data stream analyzers can presently handle.

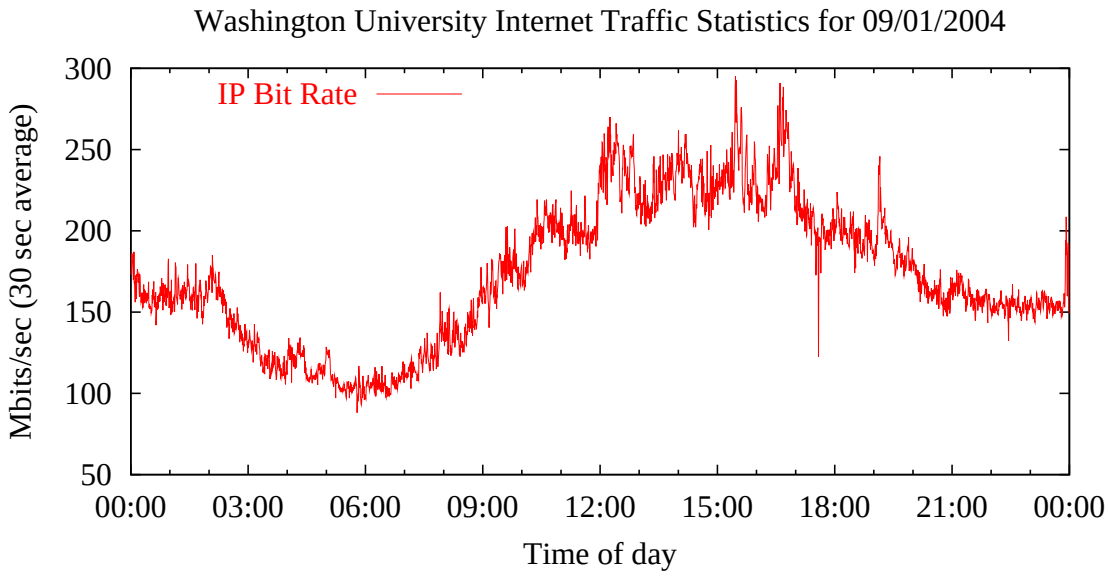


Figure 9.3: IP Packet Data Rate

The remainder of this section takes a closer look at some of the more interesting traffic patterns detected with the TCP processing hardware circuits. The TCP-Processor detected several denial of service attacks and other unexpected traffic patterns during live Internet traffic processing. Additionally, the flow scanning circuits detected the presence of worms, viruses and spam throughout the collection period. The TCP processing circuits detected denial of service attacks and virus signatures in network traffic at data rates above what existing software-based TCP flow processors can operate. These circuits also analyzed various TCP packet types in order to provide a better understanding of Internet traffic patterns. This knowledge will aid in the design of more efficient data processing systems in the future. Finally, the tests produced HTTP (port 80) and NNTP (port 119) traffic profiles which illustrate how some traffic closely follows general network usage patterns and the other traffic is time invariant.

The TCP processing circuits make no distinction is made between inbound and outbound. It is therefore impossible to differentiate between attacks originating within the Washington University network and attacks directed at the campus network. This type of information could be gathered by either monitoring inbound and outbound traffic separately, or by making modifications to the processing circuits.

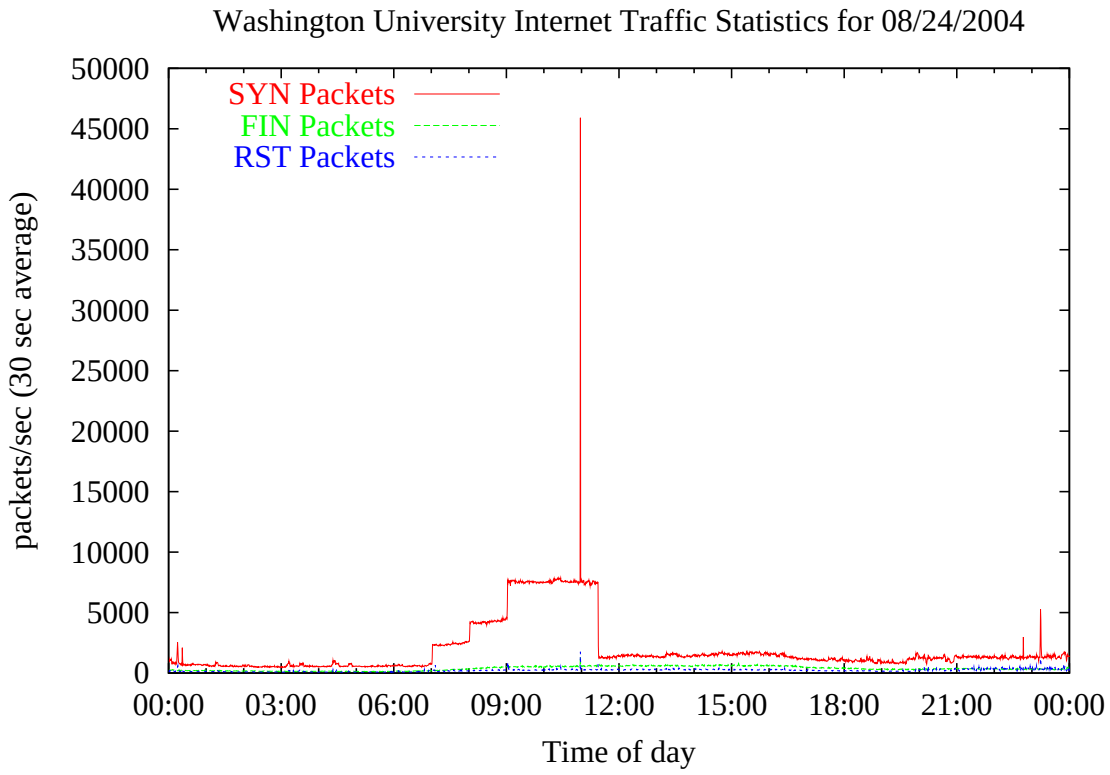


Figure 9.4: TCP SYN Flood DoS Attack

9.3.1 Denial of Service Attack

On August 28th, the TCP-Processor detected a TCP SYN denial of service attack on the Washington University network and continued to process traffic normally throughout the attack, as illustrated by the graph in Figure 9.4. The normal traffic pattern for this network averages from 1,500 to 2,000 TCP SYN packets every second. This attack had a peak traffic rate of 45,000 TCP SYN packets per second for a 30 second interval. The capture interval of the collection is 30 seconds, which implies that the duration of the burst was likely less than 30 seconds because there was no spill over of the burst into the next collection interval. If all the packets associated with this attack occurred within a one second burst, then the peak data rate would be 1.3 million packets per second.

The test uncovered several unusual characteristics of this attack. First of all, there was a noticeable hourly stepwise increase in TCP SYN activity starting several hours prior to the main attack. These increases in traffic suggest that this was a coordinated attack triggered by time, with machines in different time zones initiating the attack at one hour intervals. This stepwise increase in traffic resulted in sustained TCP SYN traffic of 8,000

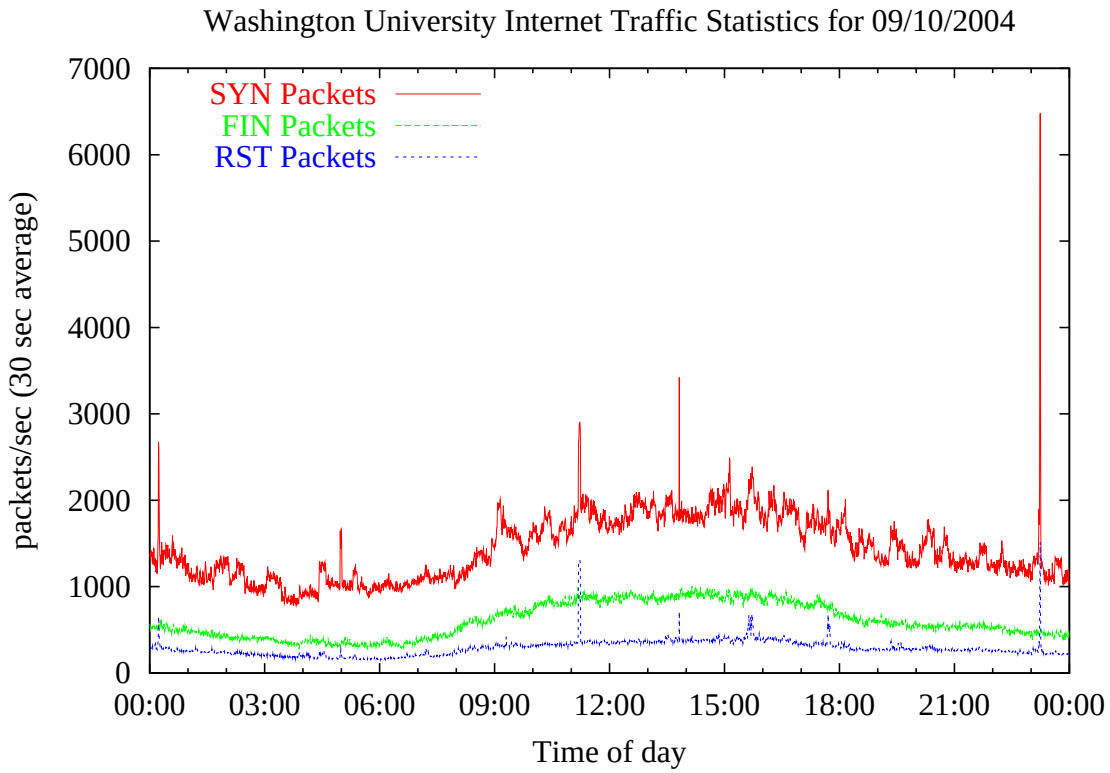


Figure 9.5: TCP SYN Burst

TCP SYN packets per second. Another interesting characteristic is the short duration of the attack. In addition to the hourly ramp-up in TCP SYN packets, there was a single burst of packets which lasted less than 30 seconds.

Another noteworthy result of the analysis was the detection of a daily burst of TCP SYN packets which occurred every evening at 11:15pm. The burst was short lived and the peak SYN rate was approximately 5000 packets per second for the 30 second collection interval. In addition to the burst of TCP SYN packets, a burst of TCP RST packets is present at the same time. Figure 9.5 shows a daily chart with a noticeable spike of TCP SYN traffic at the end of the day. This daily attack behavior can result from computers infected with viruses which perform a periodic, time-based attack. It is also possibly an artifact of some kind of programmed maintenance, such as backups or updates to virus filter definitions. The TCP-Processor provides a platform for performing anomaly detection within multi-gigabit networks.

9.3.2 Virus Detection

When used in conjunction with a content scanning hardware circuit, the TCP-Processor enables the detection of Internet worms and viruses on high-speed network links. To demonstrate this capability, the content scanning circuit was configured to search for two frequently occurring viruses, Mydoom and Netsky.

The Mydoom virus, or W32/Mydoom.b@MM virus relies on email attachments and peer-to-peer networks to spread from machine to machine. The virus infects Microsoft Windows-based machines, overwriting the host file, opening up a back door, attempting a denial of service attack, and further propagating itself. The denial of service against either www.microsoft.com or www.sco.com fails to start most of the time due to a bug in the virus itself.

The virus spreads primarily via email attachment and contains a Simple Mail Transfer Protocol (SMTP) engine which randomly selects from a list of subjects and bodies in order to trick the recipient into executing the attachment. One of the possible bodies contains the text "*The message contains Unicode characters and has been sent as a binary attachment*." The Scan circuit was configured to scan for this text. Figure 9.6 shows the occurrence of this signature over a 24 hour period. The Scan circuit detected the virus propagating at a rate of approximately five times every 10 minutes, with a peak of 15 detections per 10 minutes. Figure 9.6 shows a time invariant distribution of detections for the Mydoom virus.

The Netsky virus, or W32.Netsky.P@mm virus, is a mass mailing worm which infects Microsoft Windows-based systems. The virus is a memory resident program which exploits a well known vulnerability in Microsoft Internet Explorer. It propagates via email using the Simple Mail Transfer Protocol (SMTP) and using Kazaa network shares. The Scan circuit detected the virus by searching TCP data streams for the string "*No Virus found*" which occurs in the body of the email used to spread the virus. While it is likely that the majority of the detections were due to the Netsky virus, false positives are possible considering the relatively short ASCII text string used as the digital signature.

Figure 9.7 shows a graph of the rate at which the Netsky virus was detected. The circuit averaged approximately 12 detections per 10 minute window with peaks of 22 occurrences per 10 minute window. The virus transmissions were spread randomly throughout the 24 hour period of the chart. The time invariant distribution of detections is common for self-propagating viruses. This chart represents of the Netsky virus traffic observed over the entire five week collection period.

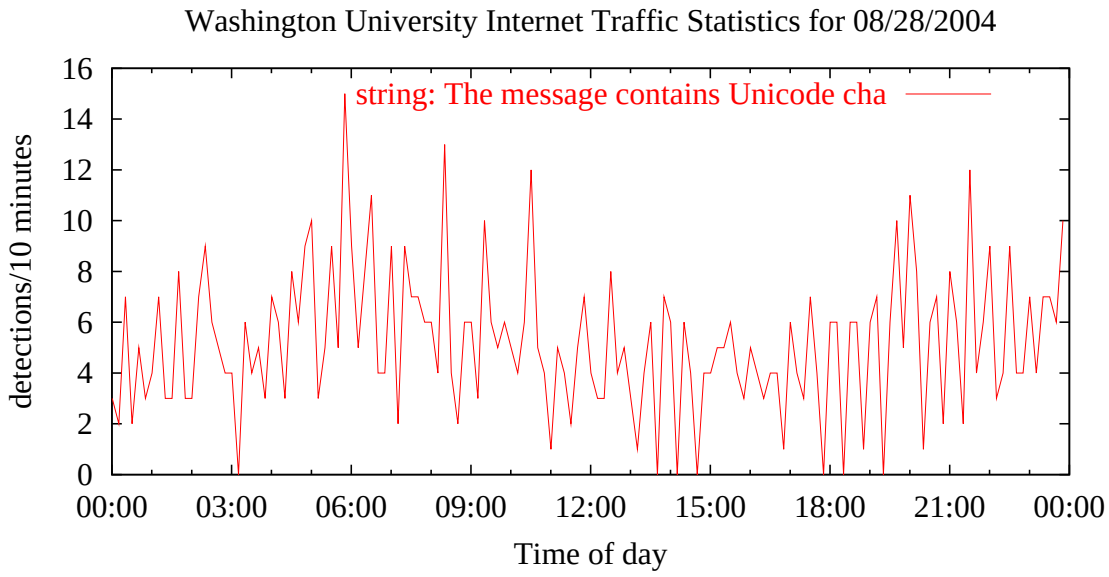


Figure 9.6: MyDoom Virus Detection

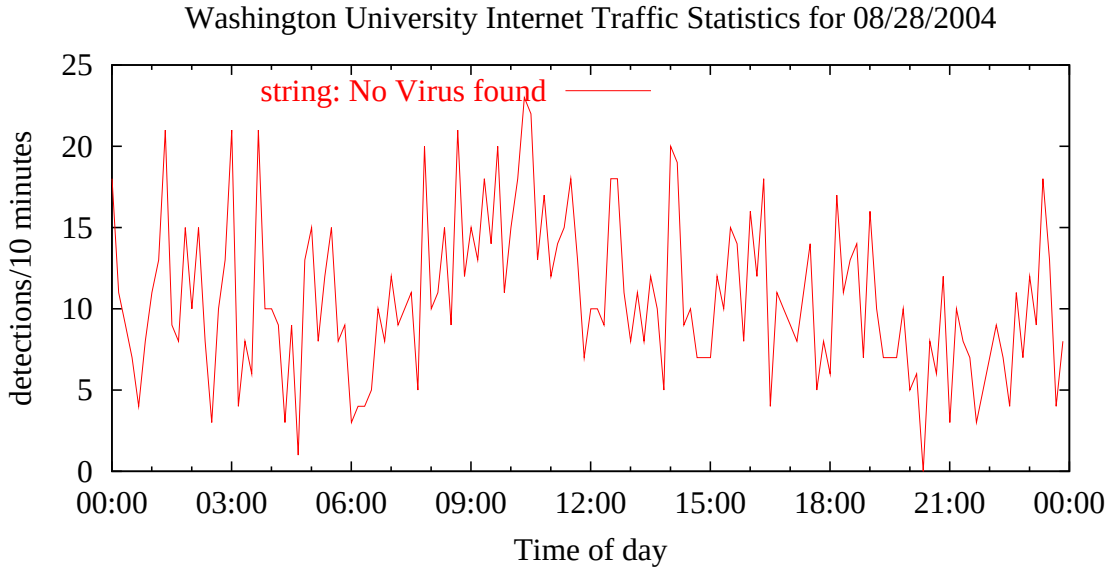


Figure 9.7: Netsky Virus

9.3.3 Spam

The TCP-Processor and associated content scanning circuits were also configured to search for spam. To demonstration this capability, they scanned all TCP data streams for the word

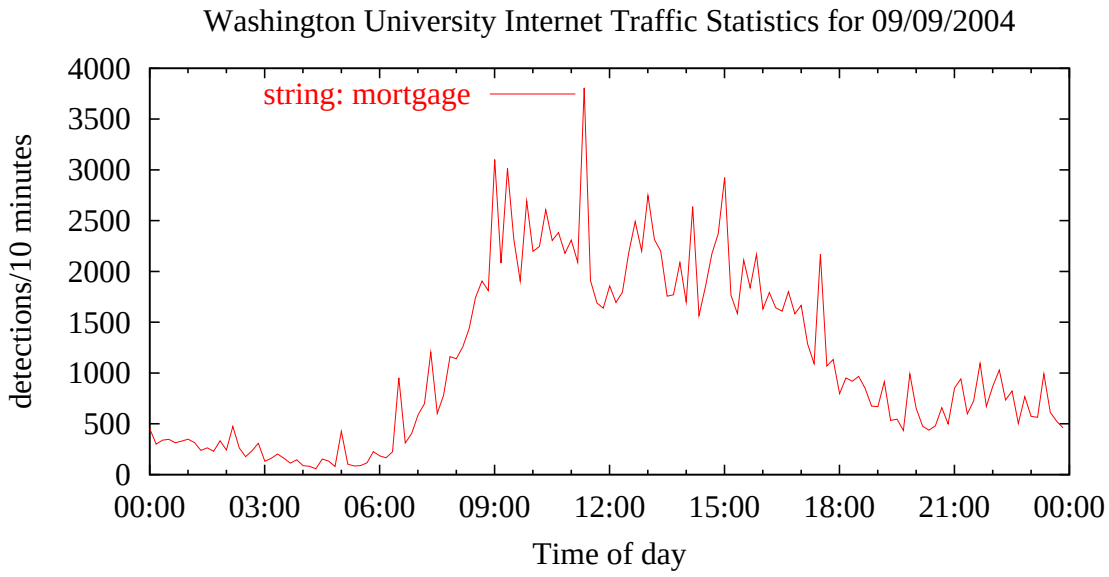


Figure 9.8: Occurrences of the String ”mortgage”

mortgage. Figure 9.8 displays a number of string detections that occurred in a 24 hour period.

The Scan circuit detected the string approximately 2,000 times per 10 minute window during the middle of the day with a peak detection rate of over 3,500 occurrences per 10 minute window. The rate at which the string was detected roughly corresponds to the overall traffic bandwidth throughout the day. The chart shows that the detection rate has the same pattern as the traffic rate shown in Figure 9.3. The spam detection results are likely skewed due to the occurrence of the string *mortgage* appearing in advertisements on web pages. Based on the data collected (see Appendix D), Web traffic constitutes 50 times more packets than email traffic and therefore will have a larger contribution to the chart.

9.3.4 Traffic Trends

Traffic patterns revealed as part of this research provide important insights into the nature of Internet traffic. These insights can now be used to improve the operation and performance of network switches, routers, and the TCP-Processor. The results have also been compared with Internet traffic analysis performed by other researchers in order to validate the operation of the TCP-Processor.

The data collected for this research confirms the findings of [112] which state that over 85% of all IP packets on the Internet are TCP packets. Figure 9.9 shows a graph

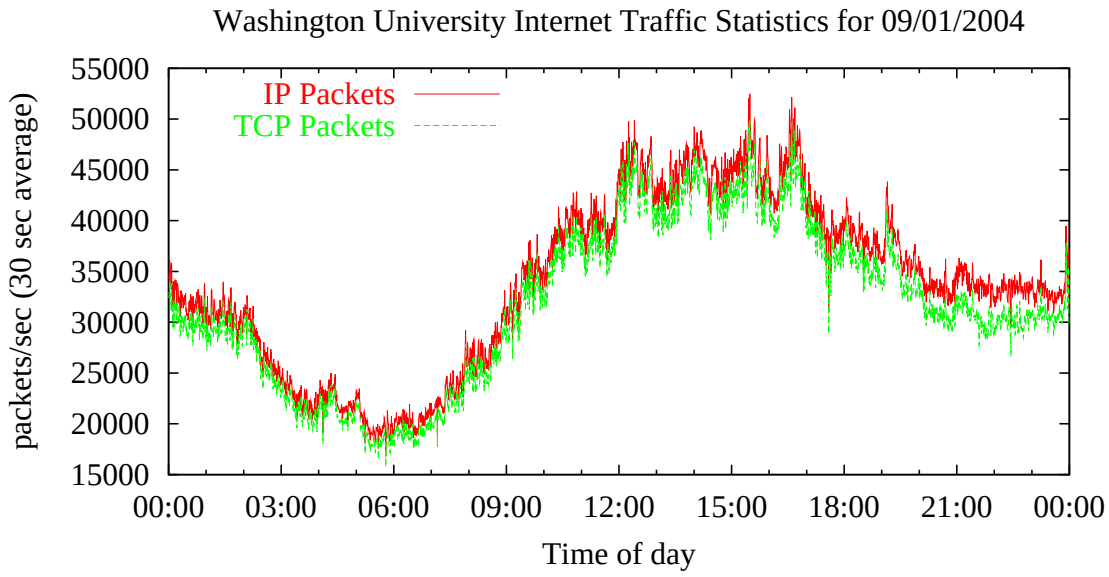


Figure 9.9: IP and TCP Packets

of both IP and TCP packets transmitted between Washington University campus network and the Internet for a 24 hour period. Throughout the day there was a consistent number of non-TCP packets while TCP traffic was subject to traffic fluctuations induced by usage behavior.

The traffic analysis also shows that 20% to 50% of all TCP packets are zero length acknowledgment packets, which supports similar findings in [112]. This statistic adds credibility to the TCP-Processor configuration option which skips TCP flow classification and per-flow context retrieval operations when processing zero-length TCP acknowledgment packets. Zero length packets are short packets (typically 40 bytes) which cause processing delays for the TCP-Processor because the amount of time required to retrieve per-flow context information is longer than the transmission time of the packet. Processing throughput of the TCP-Processor is improved when the classification of these packets is disabled. Since there is no data contained within these packets, TCP stream processing is not affected when this optimization is enabled. Figure 9.10 shows a chart comparing the number of zero-length packets to all TCP packets in a 24 hour period.

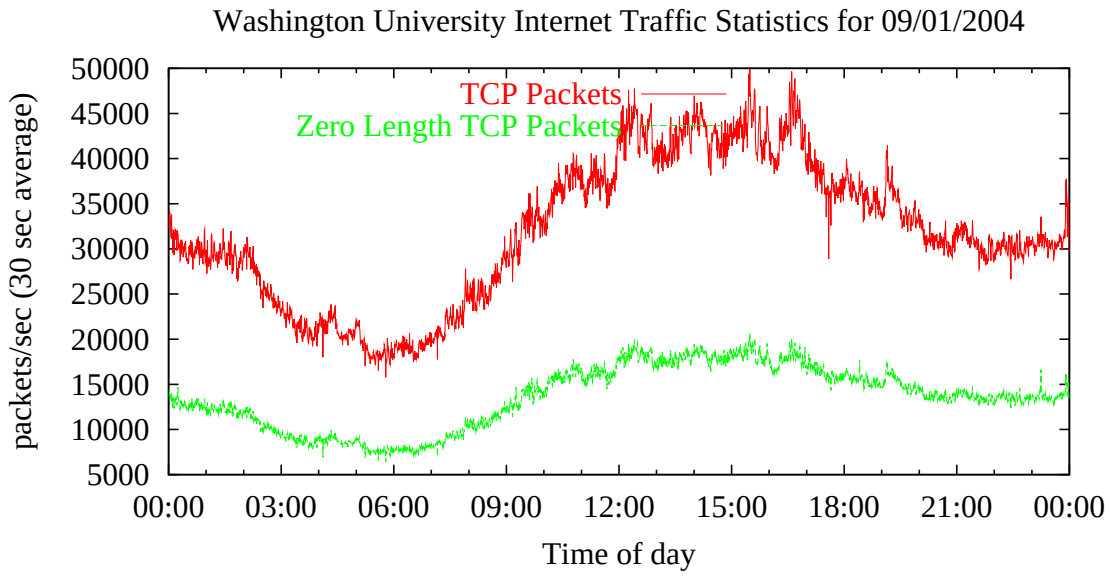


Figure 9.10: Zero Length TCP Packets

9.3.5 TCP Flow Classification

One of the challenges associated with high-performance TCP flow processing systems is accurately classifying TCP packets. The TCP-Processor circuit which processed the Washington University Internet traffic used a modified open-addressing hash table implementation to maintain per-flow context records. The experiment used a 64MB memory module capable of storing 1 million 64 byte context records.

Figure 9.11 represents the total number of active flows processed by the TCP-Processor. When it receives any TCP packet, the TCP-Processor creates a new context record in the state store if a record does not already exist for that TCP connection. A TCP FIN or TCP RST packet causes a context record to be placed in a disabled/unused state. TCP connections which are not properly terminated remain as active records in the state store. The TCP-Processor does not age out unused flow records in the current implementation. Instead, when hash collisions occur, the previous context record is replaced by a context record for the flow associated with the current packet. This behavior leads to the growth of the hash table over time caused by unterminated TCP connections which become stale context records in the state store. Figure 9.11 shows that throughout the day, between 500,000 and 700,000 flow context records are active in the system. Because stale records have the tendency to be forced out of the state store by new connections, the total number of active flows in the system decreases during the most active part of the day.

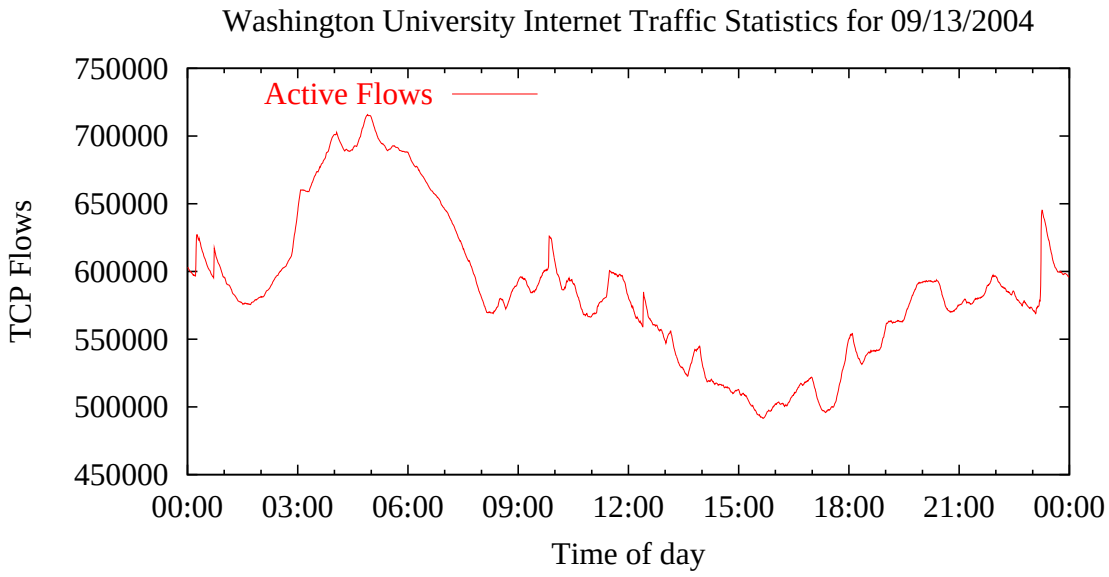


Figure 9.11: Active Flows

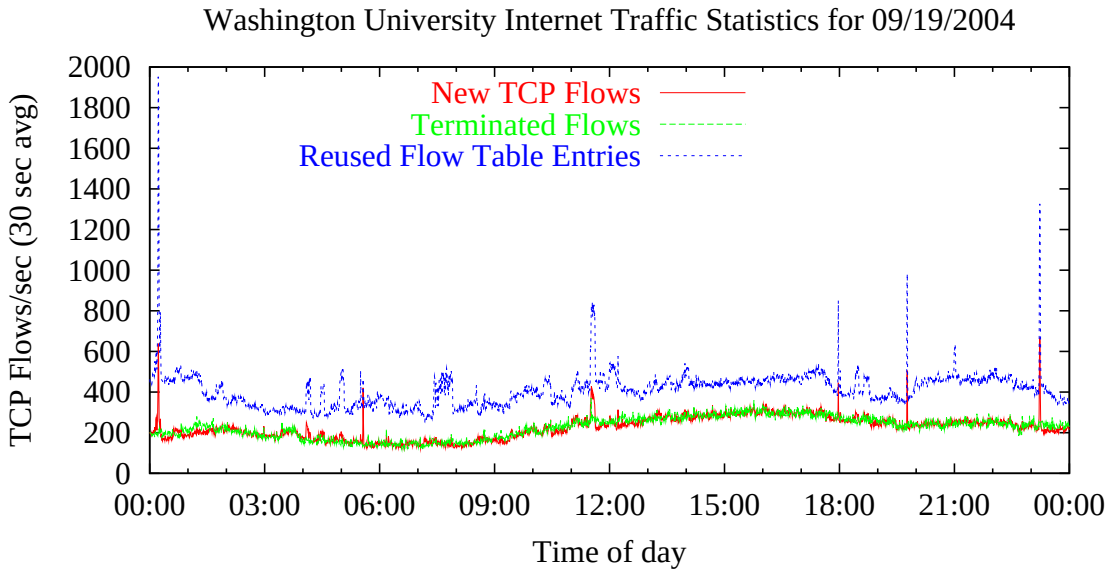


Figure 9.12: Flow Transitions

Figure 9.12 shows the number of new flow context records, terminated flow context records, and reused flow context records. The number of new TCP flows closely corresponds to the number of terminated flows. This behavior implies that the majority of the

TCP connections on the Internet are short-lived. The spikes in new flows which occur occasionally throughout the day cannot be paired with corresponding spikes in terminated flows. These spikes are likely attributable to rogue traffic such as probes to random IP addresses within the Washington University address space. If the target IP address of the probe refers to an unutilized or unreachable address in the campus network, then a TCP RST or TCP FIN packet will never be generated and the TCP-Processor will not remove the state store context for that flow.

The number of reused flow table entries is attributable to a small number of flows which continually collide with each other, causing a ping-pong effect where the presence of a packet from one flow forces the replacement of the flow context record for the other flow. The spikes in reused flow table entries are attributable to the same rogue traffic which causes bursts in the creation of new flow context records. The same traffic which causes spikes in the creation of new flows also causes spikes in reused flow entries.

9.3.6 Traffic Types

The PortTracker circuit (described on page 115) keeps track of the number of packets transiting the network which either originate from or are sent to various well known TCP ports. Two charts obtained from this data collection are presented here.

Figure 9.13 shows traffic patterns for Net News Transfer Protocol (NNTP). The NNTP packets observed on the network are invariant with respect to time. In contrast to the NNTP traffic, HyperText Transfer Protocol (HTTP) packet rates vary widely throughout the day. This variability is due to the interactive nature of HTTP Web traffic. Figure 9.14 represents HTTP traffic in a 24 hour period. This traffic pattern closely corresponds to the daily traffic pattern as shown in Figure 9.3 and Figure 9.9.

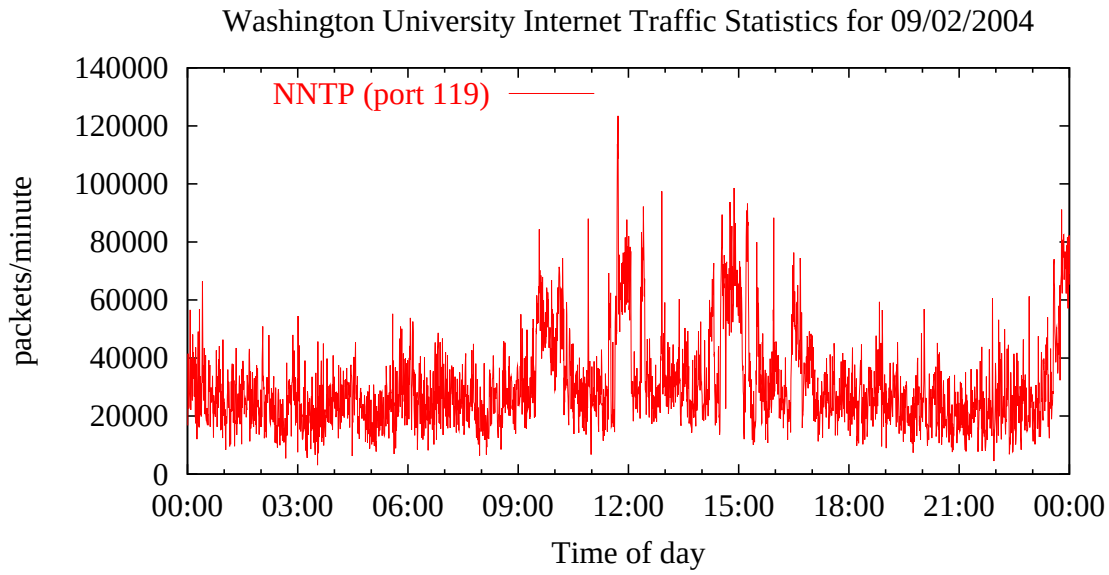


Figure 9.13: NNTP Traffic

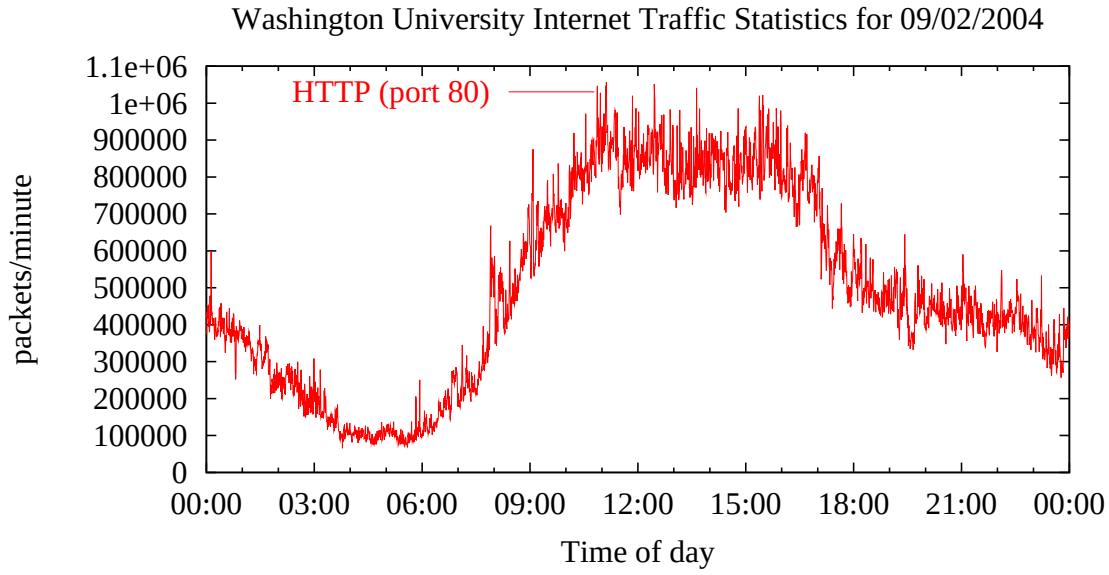


Figure 9.14: HTTP Traffic

Chapter 10

Conclusion

The TCP-Processor provides a pipelined TCP processing circuit which enables extensible networking services to process TCP flows at multi-gigabit per second data rates. It also provides stateful flow tracking, context storage, TCP stream reassembly services, and supports TCP flow manipulation for applications which operate on TCP data streams. In addition, it is capable of processing millions of active TCP connections by storing per-flow context information in an external memory module its modular design simplifies the modification and integration of components. The TCP-Processor also scales to support higher bandwidth networks by increasing parallelism, improving memory bandwidth utilization and using new generations of hardware devices which support higher clock frequencies, greater circuit densities, and increased I/O bandwidth. A serialization/deserialization module aids in the development of complex TCP flow monitoring services by providing a simple and efficient mechanism for passing annotated network packets and flow context information between multiple devices.

The completeness of the implementation is constantly at odds with the real-time processing requirements which dictate the rate at which data must be processed. The challenge is to develop a system which is stable and efficient, yet still maintains a robust feature set. The TCP-Processor succeeds in providing a real-time TCP flow processor capable of operating on high-speed network traffic.

10.1 Contributions

The TCP-Processor architecture describes a new methodology for processing TCP flows within the context of high-speed networks. By utilizing a hardware-based approach, the

architecture exploits pipelining and parallelism in order to provide an order of magnitude increase in processing performance over existing software-based TCP flow processors. Unlike other hardware-based TCP flow processors, the TCP-Processor was designed to support millions of simultaneous TCP flow contexts while processing data at multi-gigabit line rates. To accomplish this, it defines a minimal amount of per-flow context information which can be quickly swapped during the processing of packets from different flows. The per-flow context storage and retrieval operations occur in parallel with packet processing operations in order to maintain throughput at multi-gigabit per second traffic rates..

This research has resulted in the development of an architectural interface which provides a modular interface for extensible networking services. The resulting hardware interface simplifies the TCP state processing logic of the flow monitoring service. Instead of generating a separate interface specifically for handling stream content, the interface annotates fully formed network packets. It provides full access to protocol header fields to flow monitoring applications without the need for additional interface signals. In addition, since the network packets are passed through the circuit, no work is required to regenerate network packets.

A modular framework supports the integration of various plug-in modules. These modules support complex extensible networking services by providing a standard interface framework for communication between modules. Complex TCP flow monitoring services can now be developed by combining library modules with custom logic.

The TCP-Processor's flow classification and context storage subsystem provide fast access to per-flow state information required to process large numbers of flows in a single circuit design. These components perform a 96-bit exact match classification for each flow while simultaneously supporting a high rate of insertion and deletion events. This is accomplished utilizing a modified open-addressing hash scheme in which hash buckets are limited to a small fixed number of entries.

A reference implementation of the TCP-Processor protocol processing architecture uses the Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). The circuit has a post place and route frequency of 85.565 MHz when targeting the Xilinx Virtex XCV2000E-8 FPGA, utilizing 41% (7986/19200) of the slices and 70% (112/160) of the block RAMs. The device can process 2.9 million 64-byte packets per second and has a maximum throughput of 2.7Gbps.

Extensible serialization and deserialization modules support the movement of data from one device to another. These modules simplify the task of developing large and complex systems by providing a standard way to transport results between devices. The serialized data structures are self-describing and extensible so that new sections can be added and backward compatibility can be maintained. Utilizing these modules, multiple devices can be chained together to address complex network data processing requirements.

10.2 Value of Live Traffic Testing

Live traffic testing helped uncovered subtle, but critical design flaws. During the course of this research project, malformed packets were encountered which caused state transition sequences that were not originally accounted for during the design phase. In addition, other logic and processing errors in the implementation came to light during this phase of the project.

Live traffic testing also provided valuable insight into the types of anomalies one is faced with when performing protocol processing operations, especially within the area of high-speed networks. Some of the failure scenarios that were encountered involved packets with invalid header fields, runt (or short) packets, and traffic patterns which resulted in buffer overflows. One such problem involved an IP packet that had an incorrect header length field. The header length field indicated that the IP header contained 14 words (56 bytes). The length of the entire IP packet was less than 56 bytes, which caused an invalid state transition within the IP wrapper. This packet was passed into the TCP-Processor with control signals which didn't follow the interface design specification. This eventually caused the TCP-Processor to suspend processing indefinitely due to a missed control signal. Live traffic testing also helped uncover a race condition in the TCP engine which only occurred after a certain packet sequence.

Hardware circuit testing can be performed using a verification testbench or Automated Test Equipment (ATE). These systems require a set of test vectors which are applied to the inputs of the hardware circuit. The operation and output of the device under test is then compared with the expected behavior in order to determine the correctness of the circuit. The TCP-Processor performs protocol processing on network packets. The massive number of test vectors required to test all possible permutations of packet type, packet size, packet content, and packet timing make it infeasible to use a verification testbench to test more than a small number of scenarios. Live traffic testing provides at-speed processing of massive volumes of data. In addition, a ring buffer can be used to store recently

processed network packets. If a packet sequence causes problems within the circuit, the packets which triggered the processing problems can be extracted from the ring buffer and used in off-line simulations to track down and correct the problem and later be replayed through the running hardware containing recent fixes.

One of the issues associated with debugging problems in real-time systems like the TCP-Processor is the lack of visibility into the circuit when problems occur. Several techniques were used to debug the TCP-Processor while processing live Internet traffic. First, the flow control signals of the various components were tied to LEDs which would illuminate whenever the flow control signal was asserted. This provided a simple visual indication of which component stopped processing data when processing problems occurred. Debugging efforts could then be focused on a particular subcomponent as opposed to validating the operation of the whole circuit. Secondly, the circuit was instrumented with data taps at several locations throughout the circuit. This allowed network traffic to be captured and saved to external memory at various places throughout the circuit. This data could then be extracted and analyzed at a later time. By comparing data streams entering and leaving an individual circuit component, a determination could be made regarding the correctness of that component with regard to packet processing. Additionally, this captured data could be converted into simulation input files which provided the ability to trace individual logic transitions of a particular component in simulation using data sets which were causing problems in hardware.

To aid in the debugging process, a set of tools was developed which supports network traffic capture and the conversion of captured traffic patterns into simulation input files (see Appendix B on page 175). These utilities work with both standard packet capturing tools (`tcpdump`) and with custom network traffic capturing circuits. They provide a variety of data conversion and display options which simplify data analysis. Simulation input files can also be generated and used by the testbench to perform circuit simulations using ModelSim.

10.3 Utility of the TCP-Processor

The novel idea for the TCP-Processor resulted from a desire to produce a high-performance TCP processing engine which provides direct access to TCP stream content within high-speed networking environments for large numbers of flows. The TCP-Processor successfully accomplishes this goal and provides a testbed for high-performance data processing research. By allowing easy access to TCP stream content, the TCP-Processor enables other

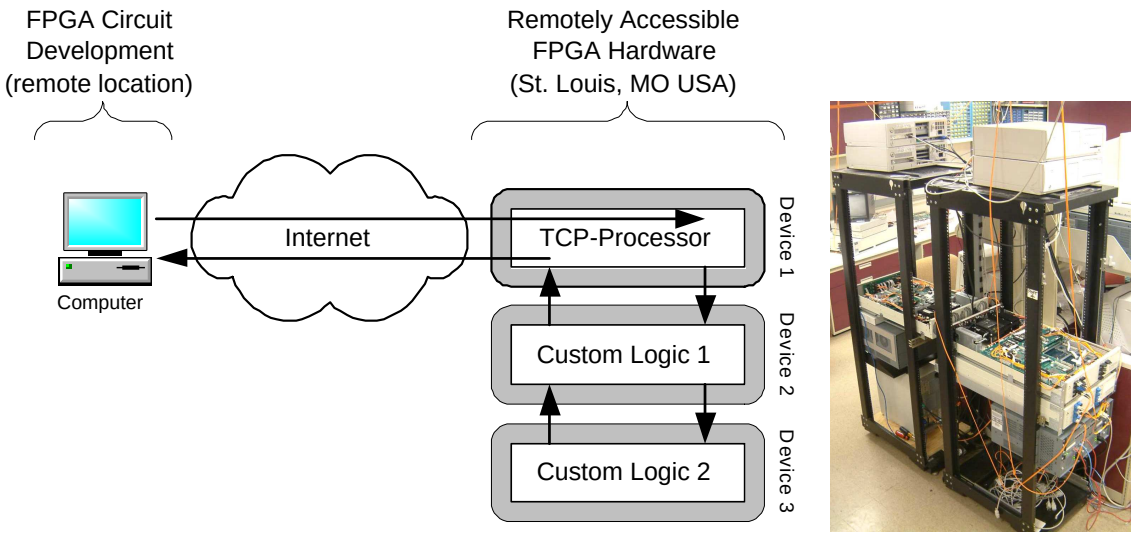


Figure 10.1: Remote Research Access

researchers to direct their attention toward developing data processing applications without being distracted by the intricacies of protocol processing.

The TCP-Processor circuit design can itself be connected to the Internet and used remotely. A remote development team can design and build a hardware circuit to perform custom data processing. That circuit design can be loaded into FPGAs at a remote location by connecting to a management station. A standard Sockets program can be written to send data to the custom processing circuits. The TCP-Processor ensures that data sent over a socket connection is delivered to the custom processing circuits. This configuration enables FPGA development for organizations which may not normally have access to these types of resources. Other networking research teams have already used this pattern of remote access for testing, specifically Emulab [136] and Open Network Lab [130, 66]. Figure 10.1 shows how a development center can be connected to the Internet at a remote location.

This remote testing environment promotes resource sharing among remote development groups. Researchers at different geographical locations can utilize the same hardware infrastructure for testing. By interleaving access to the remote hardware, the testing needs of large numbers of users can be supported by a small number of Internet connected devices. FPGA development resources can be provided to anyone that has connectivity to the Internet.

As was shown with the Grid [7], a remote testing environment provides an inexpensive mechanism for researchers to investigate FPGA-based data processing circuits for

almost any type of data set. A standard Sockets program could be developed to send data sets through a network to the remote FPGA devices. There, the TCP-Processor would extract the data set from the TCP connection and pass it to the data processing circuit. In this manner, data processing algorithms can be developed, tested, analyzed and refined without requiring the initial startup costs associated with FPGA development. If the new algorithm provides a significant improvement over existing applications, then new systems can be developed which contains custom hardware and the new processing algorithms. Possible research areas include genome sequencing, image analysis, and voice and video processing via FPGA-based digital signal processors.

In his popular 1990s book, *Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage* [121], Clifford Stoll painstakingly tracks the activities of a network intruder by monitoring characters flowing through a dial-up connection into a computer system he was managing. With the dramatic increase in programmed attacks from worms and viruses this method of detecting and tracking intrusions is extremely difficult. With the proliferation of the Internet and exponential increase in communications speeds over the past 20 years, the problem of preventing network intrusions along with limiting the spread of worms and viruses has become more difficult. In order to protect critical infrastructure sectors, such as banking and finance, insurance, chemical, oil and gas, electric, law enforcement, higher education, transportation, telecommunications, and water, the United States government put forth a National Strategy to Secure Cyberspace [132]. This document states the following:

> *"organizations that rely on networked computers systems must take proactive steps to identify and remedy their vulnerabilities"*

The TCP-Processor technology combined with intrusion detection circuits and appropriate response mechanisms can provide an excellent tool to help mitigate network attacks from cyberspace. While one technology cannot purport to solve all of the intrusion and virus problems of the Internet, steps can be taken to limit the adverse affect that this rogue network traffic has on users of the Internet. The combination of the TCP-Processor technology with header processing, content scanning, and rule matching circuits can provide an effective intrusion detection and prevention system for use in high-speed networking environments.

Chapter 11

Future Work

There are several additional research projects which can be borne out of this research. These work items include improvements to the TCP-Processor itself and also the development and enhancement of integration components and applications. This chapter describes each of these work items in more detail.

11.1 Packet Defragmentation

The Internet Protocol (IP) supports the fragmenting of packets when a single packet is larger than the Maximum Transmission Unit (MTU) size of a network segment. Efforts are made at connection setup time to avoid fragmented packets by negotiating a MTU size for the connection. While this negotiation process works well most of the time, situations do arise where a packets get fragmented within the network. The Internet traffic captured in conjunction with this research shows that roughly .3% of the IP packets were fragments. While this is a small fraction of the overall traffic, these packets cannot be ignored by a TCP flow processor. Since packet fragmentation occurs at the IP layer, defragmentation of these packets must occur prior to TCP processing.

The IP Wrapper circuit incorporated into the TCP-Processor currently does not reassemble fragmented IP packets. Enhancements are required to this circuit in order to reassemble packet fragments into the original fully formed packet. A circuit capable of defragmenting packets contains the following components:

- Logic to maintain a lookaside list of current packet fragments which are in the process of being reassembled.
- Memory to store packet fragments and retrieve them as necessary.

- Memory management logic which is capable of aging out older packet fragments or reclaiming memory from packet fragments which were never fully reassembled.
- Logic to reassemble packets after the receipt of all fragment segments and inject the newly reassembled packet into the data path.

Packet defragmentation must occur prior to the processing of the TCP protocol, which occurs in the TCP Processing Engine module (ref. Figure 4.4). The defragment component can either be added to the IPWrapper or to the Input Buffer component of the TCP-Processor. If implemented in the Input Buffer, common packet storage logic could be shared with a Packet Reordering component.

11.2 Flow Classification and Flow Aging

An extension to the TCP-Processor technology involves integrating more sophisticated packet classification algorithms. The existing flow classification algorithm uses a modified open addressing hashing technique with support for limited sized hash buckets. The TCP-Processor technology can also be used with more refined algorithms for hashing the 4-tuple of source IP address, destination IP address, source TCP port and destination TCP port. The flexibility of the TCP-Processor design simplifies the task of integrating alternative flow classification algorithms, such as trie-based, TCAM-based, or multidimensional cutting algorithms.

The open addressing hashing technique using small fixed sized hash buckets is very efficient with respect to processing time, but hash collisions during flow classification and context retrieval lead to the incomplete monitoring of flows. As shown in the Results Section on Page 141, without flow aging, the context storage area fills up over time which leads to a situation where the state store operates with most of the context area occupied. This in turn increases the probability of a hash collisions. The implementation of an age-out algorithm can improve this behavior. A flow aging feature would mark records with the time of the last activity on a particular flow. If a hash collision occurred with record which had not experienced any activity for a predefined duration, then the old record would be aged out and the new record could use vacated storage. This leads to more accurate statistics involving the number of stale records left in the state store and the true number of hash collisions among active flows.

11.3 Packet Storage Manager

The TCP-Processor currently supports two modes of packet processing. The first tracks TCP packets associated with the highest sequence number. If packets arrive out of sequence, then the stream monitoring application experiences a break in stream continuity. The second mode actively drops out-of-sequence packets in order to ensure that all packets are processed in sequence. To provide robust flow processing, a third mode of operation needs to be supported which incorporates a packet storage manager into the TCP-Processor. This packet storage manager would store out-of-sequence packets and re-injecting them into the TCP flow processing circuit in the proper sequence.

The operation of the packet store manager needs to be tightly integrated into the operation of the TCP processing engine. Information regarding where an individual packet fits within the TCP data stream is determined by the TCP processing engine. Additionally, context information relating to the current TCP stream sequence number is maintained by the TCP processing engine and stored by the state store manager. A packet store manager must communicate with these components in order to properly resequence packets.

Figure 11.1 shows the data flow between a Packet Store Manager and the TCP Processing Engine. Upon detection of an out-of-sequence packet, the TCP State Processing Logic flags the packet and stores the starting sequence number of the packet with the per-flow context. The Output State Machine routes the packet to the Packet Store Manager where it is saved in off-chip memory. When the packet arrives which advances the current sequence number to that of the stored out-of-sequence packet, the TCP State Processing Engine sends a signal to the Packet Store manager indicating that the previously stored packet should be re-injected into the data path. The Packet Store Manager retrieves the packet from off-chip memory and passes it to the TCP Engine for processing by Input State Machine and the rest of the components.

11.4 10Gbps and 40Gbps Data Rates

The TCP-Processor was designed to scale and process traffic on higher speed networks. Work is underway at the Applied Research Laboratory to develop the next generation network switch research platform [131]. This new switch will process data at 10Gbps and support advanced data processing features using the latest in FPGA technology. TCP-Processor technology is designed to scale with advances in chip technology. Utilizing current generation FPGA and memory devices, the TCP-Processor can process data at over

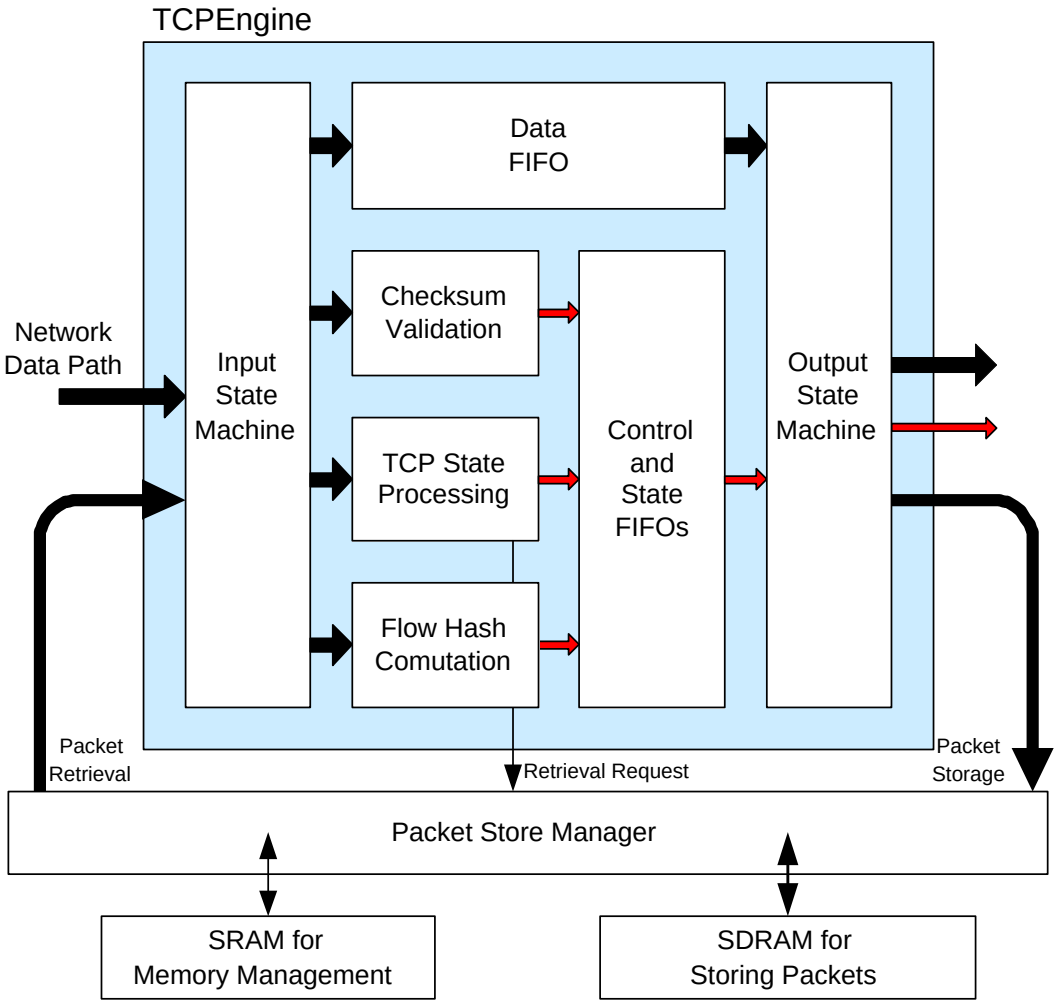


Figure 11.1: Packet Store Manager Integration

10Gbps on this platform. Additional hardware resources allow for larger and more complex flow processing applications to be supported when compared with the current FPX platform.

Further performance improvements can be obtained by migrating to the latest FPGA and memory devices. The Xilinx Virtex-4 FX140 FPGA has three times the number of logic cells as the Xilinx XCV2000E FPGA used on the FPX platform. Larger quantities of logic cells enables the TCP-Processor to employ wider bus widths and exploit higher degrees of parallelism. In addition, the latest FPGA devices can operate at 400MHz clock frequencies. Double data rate (DDR) SDRAM and quad data rate (QDR) SRAM memory devices are currently available which support data transfers on both edges of the clock, thus

doubling the available memory bandwidth. The performance of the TCP-Processor is currently limited by memory latency when processing packets less than 100 bytes in length. The increased memory bandwidth of DDR memories will likely decrease the latency associated with retrieving per-flow context data. More significant performance gains can be achieved by interleaving memory requests to improve memory bandwidth utilization. When all of these performance enhancements are combined (see Table 11.1), a version of the TCP-Processor can be constructed which is capable of processing data at OC-768 data rates.

Table 11.1: Performance improvements and estimated data rates

Improvement	Throughput for 64byte packets	
Virtex 2000E (85MHz)	1.5 Gbps	2.9 Mpps
Virtex2 8000 (167MHz)	2.9 Gbps	5.7 Mpps
Virtex4 FX140 (300MHz)	5.3 Gbps	10.3 Mpps
Parallel TCP processing engines (interleaved memory requests)	10.7 Gbps	20.9 Mpps
Two DDR memory devices (quad data paths)	21.2 Gbps	41.8 Mpps
Two FPGA devices	42.8 Gbps	83.5 Mpps

11.5 Rate Detection

Analysis of the live Internet traffic has prompted the inclusion of additional work items. The TCP-Processor currently processes TCP packets and collects statistics which are periodically broadcast to an external collection machine. Analysis of the collected statistical data occurs at a later time, sometimes days after the data was collected. In order to improve reaction times, a series of circuits needs to be developed which is capable of reacting in real-time to changes in network traffic.

The first of these items involves the generation of a rate detection circuit. This circuit would monitor the statistical event counters and test event rates against a threshold. The rate detection circuit should be flexible enough to allow for runtime modification of

the events which are monitored and of the event rate threshold values. For example, in a TCP SYN denial of service attack, the rate detection circuit would detect the burst of TCP SYN packets on the network when the TCP SYN count exceeded a predefined threshold set by a network administrator. The output of the rate detection circuit would be a signal to the appropriate response logic or the generation of an alarm.

An advanced rate detection circuit could implement more complex threshold logic which would maintain a moving average of the current traffic pattern and trigger when the current event rate exceeded the moving average by some percentage. Other complex rate detection logic could remember periodic traffic patterns, like that shown in Figure 9.3, where more traffic is expected during the middle of the day and less traffic is expected between 3:00am and 9:00am. Thresholds could then be adjusted every hour corresponding to the expected traffic pattern for that time period . This would provide more realistic event detection as opposed to setting a maximum threshold value or high water mark.

11.6 Traffic Sampling and Analysis

The ability to save network packets during periods of abnormal network behavior has significant benefit. Many network events are short-lived and by the time a network administrator notices that something is amiss, the event has passed and traffic patterns have returned to normal. Without the presence of an automated traffic capturing device, it is difficult to reconstruct the events which transpired to cause a network outage. It is not feasible to continuously capture network traffic 100% of the time. In order to gain a better understanding of a particular network attack, the capturing of network traffic must occur during the attack.

The previously mentioned rate detection circuit is ideally suited to signal a data capture circuit to collect the next million bytes of network traffic and store them in memory, typically a high-speed SRAM device. An alternative sampling methodology would be to perform a random or periodic sampling of network traffic. This type of capture would provide a broader selection of network packets over a larger time interval. Captured data could then be moved to a long term storage device or be sent to a network administrator for analysis. By analyzing exact packet sequences and timing data, information relating to the source and cause of a malicious attack can be gathered. This information may be useful in preventing future attacks.

A real-time data analysis circuit is the next logical companion to the packet capture circuit. After packets are collected in response to an exceeded threshold, an analysis circuit could then perform complex processing and interrogate the packet sequences to try and

detect patterns. For instance, sequential probing of network addresses by an attacker could be discerned from the sampled network traffic. A denial of service attack against a single IP address could also be detected by a traffic analysis circuit. Sophisticated analysis circuits could coordinate information collected at multiple locations in the network in order to produce a broader pictured of the attack profile. Analysis results could also be reported to a central monitoring station.

The final component of the real-time analysis circuit would be a feedback mechanism which could alter the behavior of networking equipment based on the results of the analysis. The TCP-Processor could be enhanced with circuitry capable of rate limiting TCP SYN packets or drop packets based on some criteria. During a denial of service attack, the rate detection circuit would detect an increase in traffic and signal the sampling circuit to capture network packets. After storing a subset of the network traffic, the analysis engine would begin scanning the saved packets looking for patterns. In this particular example, a TCP SYN attack is detected which originates from a single source IP address. The analysis circuit would then inform the TCP-Processing logic to rate limit TCP SYN packets from the offending host. In addition, the analysis circuit would send an alert message to a network administrator summarizing the current attack and the response action taken. This sequence of events describes and automated response system capable of limiting the impact of a network attack on an organization or network segment. The collection of these enhancements would allow for in-depth analysis of network events and support real-time responses to rogue network traffic.

11.7 Application Integration

The final work item that stems from this research is the development and integration of other flow processing applications. These applications would connect to the TCP-Processor to received TCP stream content and perform custom processing on the data. Due to the in-band processing design of the TCP-Processor, these applications can also generate TCP packets and alter TCP stream content.

A class of TCP processing applications which perform intrusion detection and prevention functions are ideally suited for use with the TCP-Processor. These applications perform rules-based content scanning and protocol header filtering in order to classify packets as rogue traffic. Upon detection of a computer virus, an intrusion detection system would note the presence of the virus and let it pass through the network. An intrusion prevention system would attempt to halt the spread of the virus. This can be accomplished by dropping

the packet from the network and then terminating the TCP connection by generating TCP RST or TCP FIN packets. A more sophisticated approach would be to disarm the virus by altering or removing the virus from the the TCP data stream.

With the adoption of Internet SCSI (iSCSI) and the advent of network attached storage (NAS) devices where data is accessed by communicating with NAS using the SCSI protocol encapsulated in a TCP connection. The wide acceptance of this type of storage mechanism implies that more and more data will be accessed through a network using the TCP protocol. Applications based on the TCP-Processor technology will be able to process the abundance of data stored on these devices at performance levels unattainable using microprocessor based systems.

The amount of traffic on the Internet and the number of systems connected to the Internet continues to grow at a rapid pace. The use of TCP Offload Engines (TOEs) will only increase the amount of traffic on networks and speed at which data is pushed through a network. In the future, applications based on the TCP-Processor or similar technology will be required to effectively monitor and process TCP data within a network.

Appendix A

Usage

This section describes in detail how to use the TCP-Processor circuit. It provides information on how to compile, simulate, synthesize, place & route, and run the various TCP-Processing circuits. This documentation assumes that the user is familiar with ModelSim, Synplicity, Xilinx tools, cygwin, gcc, and a text editor for modifying files. These circuits were developed utilizing ModelSim 5.8SE, Synplicity 7.2Pro, and Xilinx ISE 6.1i.

Figure A.1A shows a logical diagram of the encoding and decoding operations as traffic is passed among the StreamExtract, the PortTracker, and the Scan circuits. Figure A.1B shows a physical layout of how the FPX devices can be stacked in order to match the logical layout.

The circuits easily achieve a post place & route clock frequency of 55 MHz targeting a Xilinx Virtex XCV2000E-6. A maximum clock frequency of 87 MHz has been achieved for the StreamExtract circuit when targeting the -8 speed grade part.

A.1 StreamExtract

The distribution for the StreamExtract circuit can be found at the URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/source/streamextract_source_v2.zip`. To install, unzip the distribution into an appropriately named directory. Edit the file `vhdl/streamextract_module.vhd` and modify the configuration parameters as required by the target environment. See the Configuration Parameters section on Page 76 for a full discussion on the various configuration options and the effect that they have on the operation of the circuit.

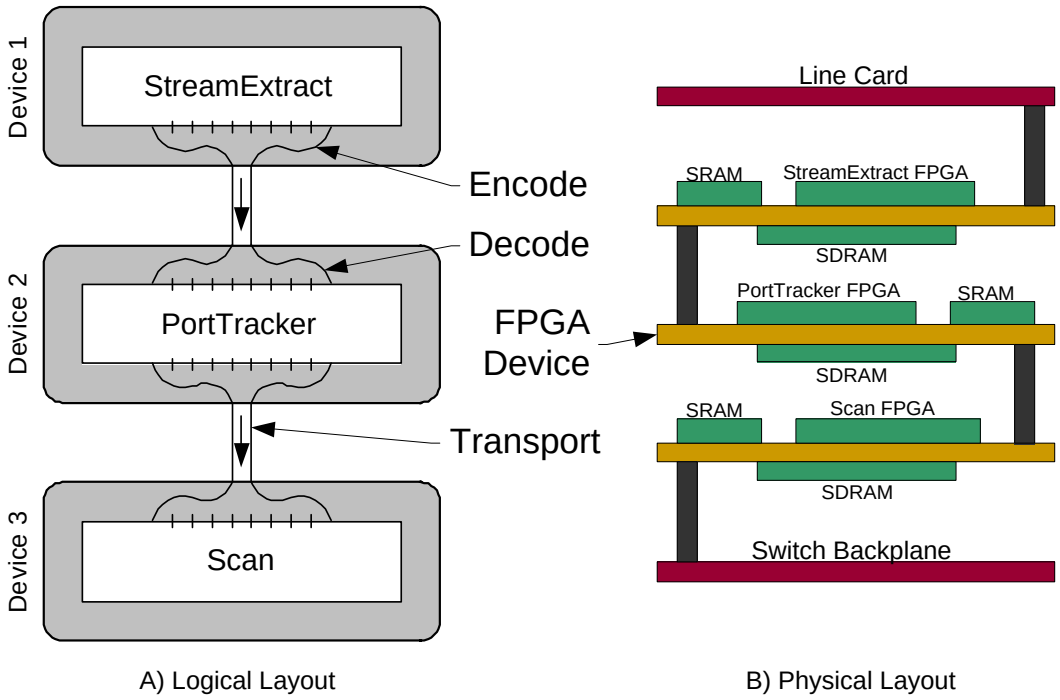


Figure A.1: Multidevice Circuit Layout

A.1.1 Compile

The compilation of the VHDL source is a very straightforward process. A Makefile compiles the code, assuming the machine has been properly setup with the aforementioned tools, environment variables, licenses, and search paths. The command is:

```
$ make compile
```

This operation should complete successfully. If errors occur, they can be resolved using the information provided in the error message.

A.1.2 Generating Simulation Input Files

Prior to simulation, generate a simulation input file. The easiest method is to use the `tcpdump` [83] command to capture network traffic, such as HTTP web requests. This can be accomplished with a laptop computer connected to a Linux machine which acts as a gateway to the Internet. A `tcpdump` command like the one below will capture the network traffic to a file.

```
$ tcpdump -i eth1 -X -s 1514 host 192.168.200.66 > tmp.dmp
```

Change the `tcpdump` parameters in order to obtain the proper network capture. Specify the `-X` flag to indicate a hexadecimal output format and the `-s 1514` parameter to capture the complete contents of each packet.

Next, convert the `tcpdump` output format into a ModelSim input file. This task can be accomplished using the `dmp2tbp` application. Information on the dmp2tbp utility along with links to the executable and source can be found at:
`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/dmp2tbp.html`.

The output of the `dmp2tbp` program is a file called INPUT_CELLS.TBP. This file is used as an input to the IPTESTBENCH [14] program. Copy the INPUT_CELLS.TBP file to the `sim` directory and run the command:

```
$ cd sim
$ make input_cell
```

This generates a ModelSim input file called INPUT_CELLS.DAT. With this file, simulations can be performed. Note that the `dmp2tbp` application inserts two header words before the start of IP header so the **NUM_PRE_HDR_WRDS** configuration parameter will need to be set to a value of 2.

A.1.3 Simulate

Once the VHDL source code is compiled, and a simulation input file is generated, it is possible to perform functional simulations. This is accomplished by entering the following command:

```
$ make sim
```

This brings up ModelSim, a Signals window, a Waveform window, and runs the circuit for 65 microseconds. If problems occur, check for errors on the command line and in the ModelSim output window. Resolve any problems and rerun the simulation. At this point, use the normal ModelSim functions to interrogate the operation of the circuit. Use the ModelSim `run` command to simulate larger input files.

The StreamExtract testbench generates a file called LC_CELLSOUT.DAT which contains encoded TCP stream data. This file can be used as the simulation input for circuits like the PortTracker and Scan application which use the TCPLiteWrappers.

A.1.4 Synthesize

Prior to synthesizing the circuit design, edit the `vhdl/streamextract_module.vhd` source file and change the configuration parameters to reflect the target environment. Most notably, change the *Simulation* parameter to zero which indicates that all of memory should be initialized and utilized during the operation of the circuit. At this point, either run Synplicity in batch mode or interactive mode. For batch mode operation, enter the following command:

```
$ make syn
```

For interactive mode, initiate execution of the Synplify Pro program, load the project file `sim/StreamExtract.prj`, and click *run*. Regardless of the mode of operation, check the `syn/streamextract/streamextract.srr` log file and look for warnings, errors, and the final operation frequency. Edit the implementation options to target the correct speed grade and clock frequency for the target device.

A.1.5 Place & Route

The place & route operations are executed in batch mode and are initiated by the following make file:

```
$ make build
```

To change the target FPGA device, edit the file `syn/rad-xcve2000-64MB/build` and change *part* variable to equal the appropriate device. To change the target clock frequency, edit the file `syn/rad-xcve2000-64MB/fpx.ucf`, go to the bottom of the document and modify the *timespec* and *offset* variables to correspond to the desired clock frequency.

The final step in the build process is the generation of the bit file. Prior to using the bit file, verify that the operational clock frequency of the newly created circuit meets or exceeds the design constraints. The timing report for the circuit can be found in the file `syn/rad-xcve2000-64MB/streamextract.twr`. If the circuit achieves the desired performance levels, load the circuit into an FPGA and start processing traffic. The bit file can be found at `syn/rad-xcve2000-64MB/streamextract.bit`.

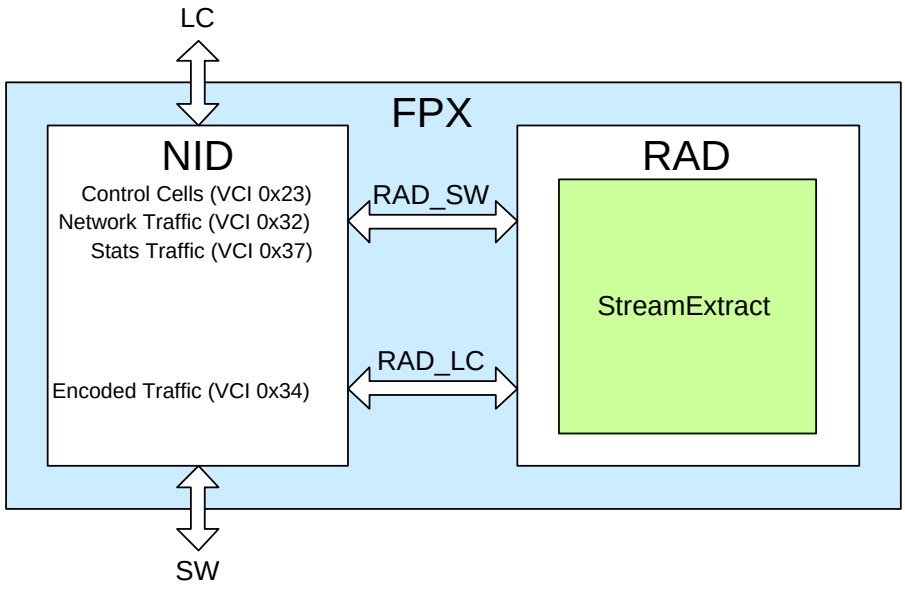


Figure A.2: StreamExtract NID Routes

A.1.6 Setup

Now that the circuit has been built, it is ready to be loaded into an FPX device. To accomplish this task, use the NCHARGE application to perform remote configuration, setup and programming of the FPX device.

Prior to sending traffic through the device, set up the NID routes as shown below. Figure A.2 shows the virtual circuit routes between the NID and the RAD for the StreamExtract circuit. Raw network traffic should be routed through the RAD_SW interface over VPI 0x0 VCI 0x32. In addition, control cell traffic (VPI 0x0 VCI 0x23) and statistic traffic generated by the StreamExtract circuit (VPI 0x0 VCI 0x37) should be routed over the RAD_SW interface. The virtual circuit used by the statistics module can be changed via the previously discussed StreamExtract configuration parameters. Encoded TCP stream data is passed through the RAD_LC interface using VPI 0x0 VCI 0x34.

A.2 Scan & PortTracker

Due to their similarity, the discussions on the Scan and PortTracker applications have been grouped together. The distribution for the Scan circuit can be found at the URL:
`http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp`

/source/scan_source.zip
The distribution for the PortTracker circuit can be found at the URL:
http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp
/source/porttracker_source.zip
Unzip the distributions into appropriately named directories. Edit the vhdl/scan_module.vhd and vhdl/porttracker_module.vhd source files and modify the configuration parameters as required by the target environment. See the previous section on configuration parameters for a full discussion on the various parameters and the effect that they have on the operation of the circuits.

A.2.1 Compile

The compilation of the VHDL source is a very straightforward process. A Makefile compiles the code. The command is shown below:

```
$ make compile
```

If errors occur, resolve the errors utilizing information provided in the error message.

A.2.2 Generating Simulation Input Files

Prior to simulation, generate a simulation input file which contains encoded TCP stream data. The simulation of the StreamExtract circuit generates the appropriately formatted simulation input file. Copy the sim/LC_CELLSOUT.DAT file from the StreamExtract project to the sim/INPUT_CELLS_LC.DAT file in the current project. A special input file can be created to test the control and stats features by utilizing the same methods as outlined in the StreamExtract project for creating an INPUT_CELLS.DAT file.

A.2.3 Simulate

After compiling the VHDL source code and generating a simulation input file, functional simulations can be performed. To accomplish this, enter the following command:

```
$ make sim
```

This brings up ModelSim, a Signals window, a Waveform window, and runs the circuit for a short period of time. Longer simulation runs can be produced by using the ModelSim run command to simulate more clock cycles.

A.2.4 Synthesize

Prior to synthesis, edit the `vhdl/scan_module.vhd` and `vhdl/porttracker_module.vhd` source files and change the configuration parameters to reflect the target environment. Most notably, change the *Simulation* parameter to zero which indicates that all of memory should be initialized and utilized during the operation of the circuit. At this point, either run Synplicity in batch mode or interactive mode. For batch mode operation, enter the following command:

```
$ make syn
```

For interactive mode, start up the Synplify Pro program, load the `sim/Scan.prj` or `sim/PortTracker.prj` project file, and click *run*. Regardless of the mode of operation, check the `syn/scan/scan.srr` or `syn/porttracker/porttracker.srr` log files and look for warnings, errors, and the final operation frequency. Edit the implementation options to target the correct speed grade and clock frequency for the target device.

A.2.5 Place & Route

The place & route operations are executed in batch mode and are initiated by the following command:

```
$ make build
```

To change the target FPGA device, edit the file `syn/rad-xcve2000-64MB/build` and change *part* variable to equal the appropriate device. To change the target clock frequency, edit the file `syn/rad-xcve2000-64MB/fpx.ucf`, go to the bottom of the document and modify the *timespec* and *offset* variables to correspond to the desired clock frequency.

The final set in the build process is the generation of the bit file. Prior to using the bit file, verify that the operational clock frequency of the newly created circuit meets or exceeds the design constraints. The timing report for the circuit can be found in the file `syn/rad-xcve2000-64MB/scan.twr` or `syn/rad-xcve2000-64MB/porttracker.twr`. If the circuit achieved the desired performance levels, load the circuit into an FPGA and start processing traffic. The bit file can be found at `syn/rad-xcve2000-64MB/scan.bit` or `syn/rad-xcve2000-64MB/porttracker.bit`, depending on which application is being built.

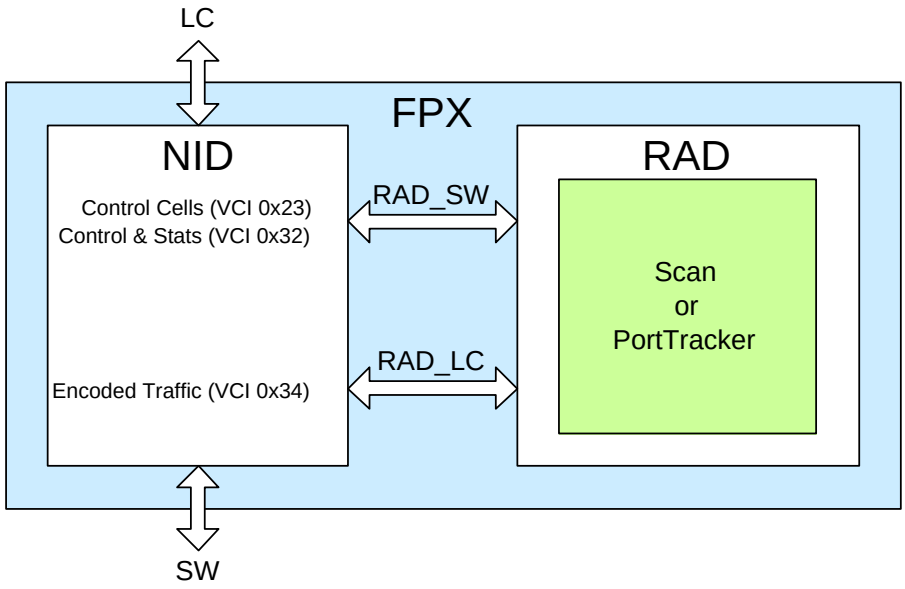


Figure A.3: Scan/PortTracker NID Routes

A.2.6 Setup

Now that the circuit has been built, it is ready to be loaded into an FPX device. The steps required to perform this task are identical to those of the StreamExtract circuit. Next, set up the NID routes. The NID routes for both the Scan and the PortTracker are the same. Figure A.3 shows the virtual circuit routes between the NID and the RAD for the Scan and PortTracker circuits. Network traffic carrying control and statistics information should be routed through the RAD_SW interface over VPI 0x0 VCI 0x32. Control cell traffic should be routed to the RAD_SW interface over VPI 0x0 VCI 0x23. Encoded TCP stream data is passed through the RAD_LC interface using VPI 0x0 VCI 0x34.

A.3 New Applications

Prior to starting development of a new TCP stream data processing application, it is important to analyze the environment, resources, and size of the target application. The StreamExtract circuit, the PortTracker circuit, and the Scan circuit are three recommended starting points for new application development. All of these projects are structured in a similar manner so that the transitions from one environment to another are easily achieved without requiring mastery of different interfaces, file structures, or build processes.

The StreamExtract circuit should be utilized if a small footprint solution is desired. This allows the TCP stream processing application to coexist in the same circuit as the TCP-Processor, thus utilizing only one device. The PortTracker circuit is a good example of a multi-device solution where the TCP-Processor operates on one device and the TCP stream processing application operates on one or more other devices. Since a separate device is utilized for the stream processing application, more FPGA resources are available to perform processing. The Scan circuit is similar to the PortTracker circuit, except it also contains additional logic to store and retrieve per-flow context information in off-chip SDRAM devices. The Scan circuit also contains logic for processing external commands formatted as UDP data packets.

Once the appropriate starting point for the intended TCP stream processing application has been determined, the selected example circuit can be downloaded and extracted in order to begin the desired modifications. If using the StreamExtract circuit as the starting point, edit the file `streamextract.vhd`. Remove the *TCPSerializeEncode* and *TCPSerializeDecode* components and replace them with components specific to the new application. If the Scan or PortTracker circuits are selected as the starting point, edit the `_module.vhd` source file and replace either the ScanApp or PortTrackerApp circuits with logic for the new application. Changes will also be required to the `controlproc.vhd` source file to support different control and statistics features.

A.3.1 Client Interface

The client interface is identical for a TCP stream processing application based on either the StreamExtract, PortTracker, or Scan circuits. Detailed information on this interface was previously described on Page 95 in the section covering the TCPRouting component of the TCP-Processor. This information covers the interface signals, a sample wave form, and details on the data passed through the interface. Another excellent approach to gaining familiarity with the client interface of the TCP-Processor (and TCPLiteWrappers) is to run simulations with various data sets and inspect the various signal transitions.

A.4 Runtime Setup

The FPX cards on which the TCP processing circuits operate can either be inserted into the WUGS-20 switch environment or the FPX-in-a-Box environment. Runtime setup and configuration information is presented for both of these environments. While the circuits

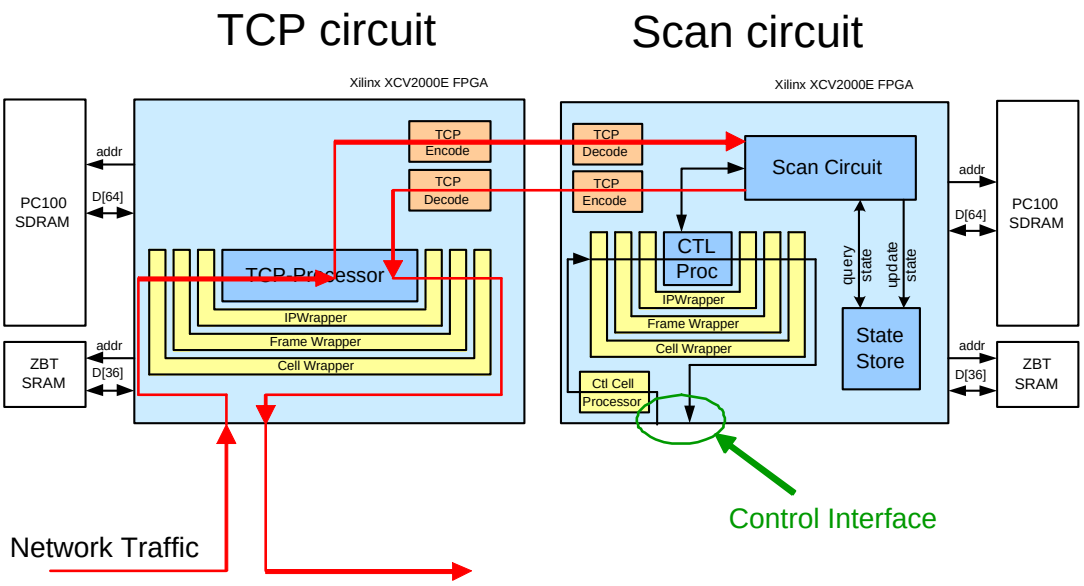


Figure A.4: Multidevice Circuit Layout

and the FPX cards are identical for both of these environments, there are configuration differences due to the WUGS-20 backplane's ability to route traffic from any port to any other port. The FPX-in-a-Box solution, conversely, relies on the NID to perform all traffic routing functions. The interaction between two FPX devices working together to provide a TCP stream monitoring service is shown in Figure A.4. Network traffic processed by the TCP-Processor is encoded and passed to a separate device for additional processing. This second device processes the encoded TCP stream data and searches for digital signatures. The encoded data is passed back to the TCP-Processor for egress processing and to forward data on to the end system or next hop router. This final step is not required when performing passive monitoring because the network packets do not need to be delivered to the end system via this network path.

Figure A.5 shows a WUGS-20 switch configured to perform active monitoring of inbound Internet traffic. A laptop computer initiates communications with hosts on the Internet. These communications can include remote logins, file transfers and web surfing. Traffic initiated on the laptop is passed through a network hub and routed to an OC-3 line card installed at Port 1 of the switch. This traffic is passed directly to the OC-3 line card at Port 0, transferred to a different network hub and routed to the Internet. Return traffic from the Internet is passed to the line card at Port 0. This network traffic is routed to an FPX card at Port 7 which is running the StreamExtract circuit. Encapsulated TCP stream data is

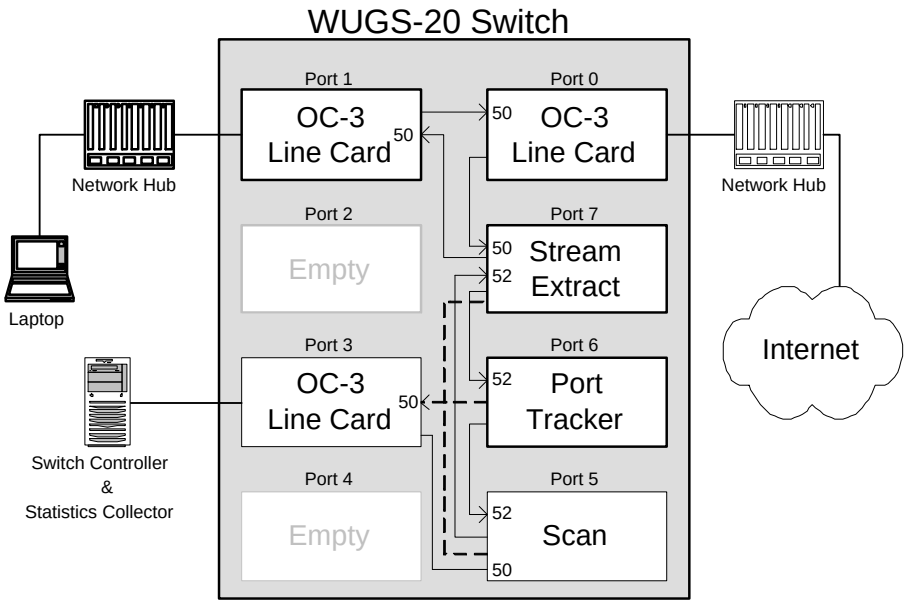


Figure A.5: Active Monitoring Switch Configuration

passed to the PortTracker circuit at Port 6 and then to the Scan circuit at Port 5. After the Scan circuit, this encoded network traffic is passed back to the StreamExtract circuit at port 7 which reproduces the original network data packets and passes them to the line card at Port 1 for delivery to the laptop. Statistics information generated by the StreamExtract, the PortTracker and the Scan circuits is passed to the OC-3 Line card at Port 3 which forwards the statistics to a host machine which collects them. The statistics traffic is shown as a dashed line in the figure.

The VCI routes which need to be configured on the WUGS-20 switch are shown in Table A.1. The table is broken into four sections showing routes used for network traffic, encoded network traffic, statistics traffic, and control traffic. The Scan circuit supports remote management features and therefore requires control traffic. NID routing information can be found in the previous sections dealing with each of the specific circuits.

Figure A.6 shows a WUGS-20 switch configured to perform passive monitoring of network traffic. Mirroring, replication or optical splitting techniques can be used to acquire copies of network traffic for passive monitoring. In this configuration, the traffic to be monitored is passed into the GigE line card at Port 1. Traffic is then passed to the StreamExtract circuit at Port 2 where TCP processing is performed. Encoded network traffic is passed via VCI 52 to the PortTracker circuit at Port 4 and the Scan circuit at Port 6. Since this is a passive monitoring configuration, there is no need to route encoded traffic

Table A.1: Routing Assignments for Active Monitoring Switch Configuration

Network Traffic
Port: 1 VCI: 50 => Port: 0 VCI: 50
Port: 0 VCI: 50 => Port: 7 VCI: 50
Port: 7 VCI: 50 => Port: 1 VCI: 50
Encoded Network Traffic
Port: 7 VCI: 52 => Port: 6 VCI: 52
Port: 6 VCI: 52 => Port: 5 VCI: 52
Port: 5 VCI: 52 => Port: 7 VCI: 52
Statistics Traffic
Port: 7 VCI: 55 => Port: 3 VCI: 50
Port: 6 VCI: 50 => Port: 3 VCI: 50
Port: 5 VCI: 50 => Port: 3 VCI: 50
Control Traffic
Port: 3 VCI: 50 => Port: 5 VCI: 50

back to the StreamExtract circuit for egress processing. Statistics information generated by the StreamExtract, the PortTracker and the Scan circuits is passed to the GigE Line card at Port 0 which forwards the statistics to the stats collection machine.

The VCI routes which need to be programmed into the WUGS-20 switch for the passive monitoring configuration are shown in Table A.2. The table is broken into four sections showing routes used for network traffic, encoded network traffic, statistics traffic, and control traffic. The Scan circuit supports remote management features and therefore requires control traffic. NID routing information can be found in the previous sections dealing with the each of the specific circuits.

The runtime setup for TCP stream processing circuits operating within a FPX-in-a-Box environment is different from that of a WUGS-20 switch environment. The main reason for this is that the backplane for the FPX-in-a-Box solution only provides an electrical signal interconnect between the two stacks of FPX devices and no routing services. This means that all routing functions must be performed by the NID circuit on each of the FPX devices.

Figure A.7 shows a configuration for performing passive monitoring utilizing a FPX-in-a-Box environment. Similar to passive monitoring configurations in a WUGS-20 switch environment, an external device is required to provide duplicated or mirrored network traffic to the input line card (located at the top of the stack on the left). Network traffic to be monitored is passed to the StreamExtract circuit for TCP processing. Encoded

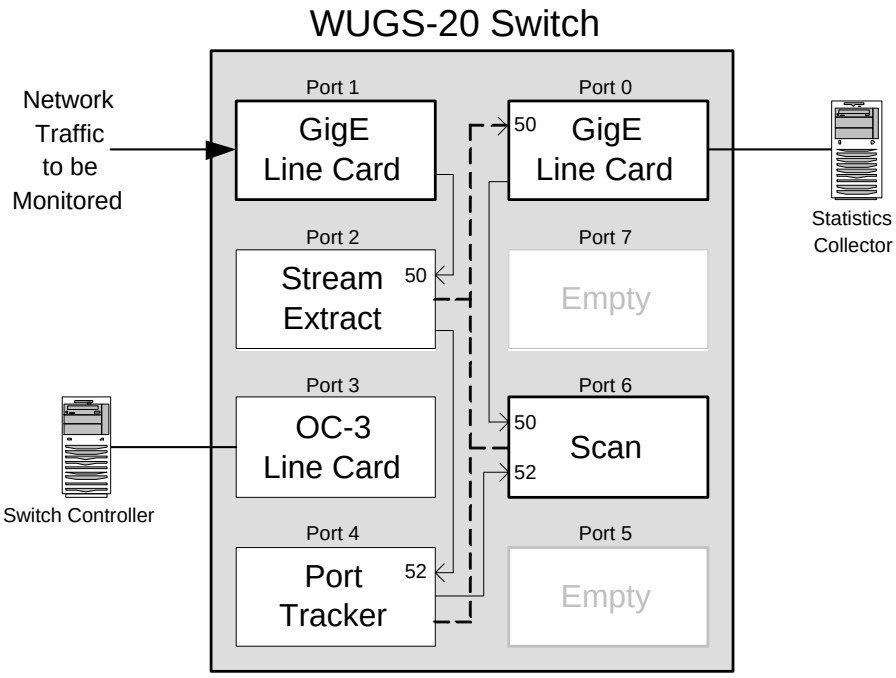


Figure A.6: Passive Monitoring Switch Configuration

Table A.2: Routing Assignments for Passive Monitoring Switch Configuration

Network Traffic
Port: 1 VCI: 51 => Port: 2 VCI: 50
Encoded Network Traffic
Port: 2 VCI: 52 => Port: 4 VCI: 52
Port: 4 VCI: 52 => Port: 6 VCI: 52
Statistics Traffic
Port: 2 VCI: 55 => Port: 0 VCI: 50
Port: 4 VCI: 50 => Port: 0 VCI: 50
Port: 6 VCI: 50 => Port: 0 VCI: 50
Control Traffic
Port: 0 VCI: 50 => Port: 6 VCI: 50

network packets are passed down to the PortTracker circuit and finally down to the Scan circuit. The NID cannot currently be configured to act as a data sink for network traffic. For this reason, a stream processing circuit (such as the Scan circuit) will have to be modified to not pass encoded network traffic out through the RAD_LC interface. Without this

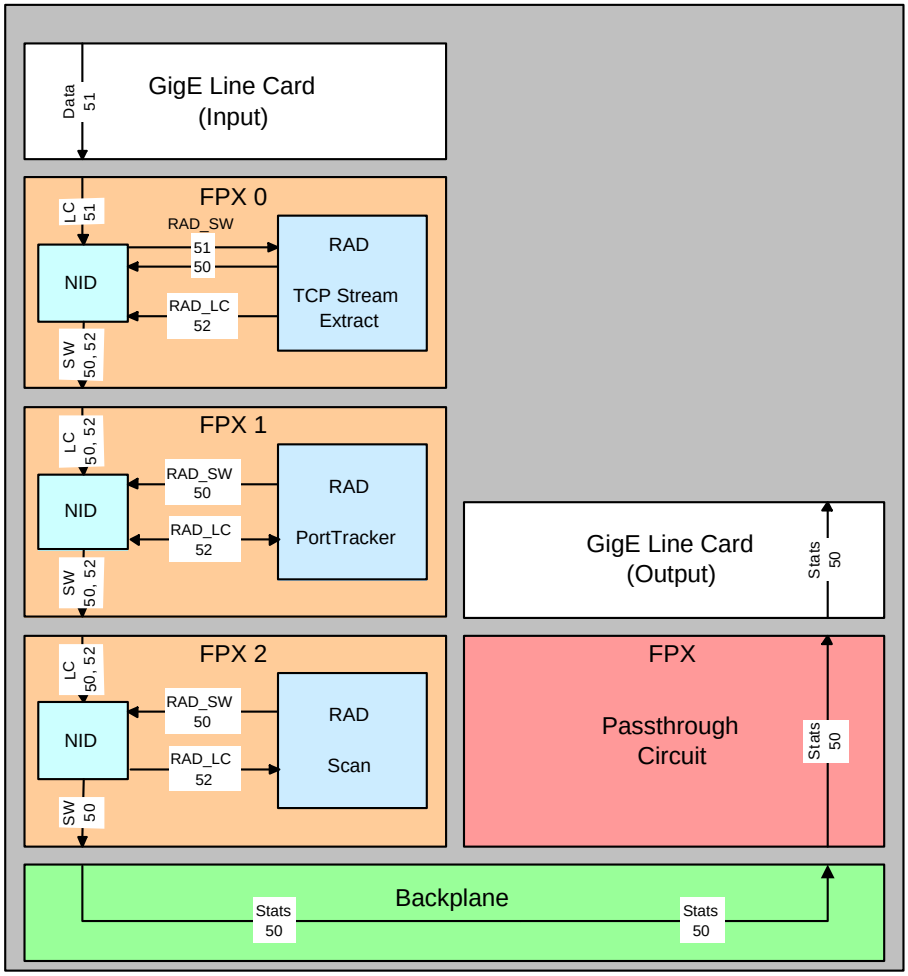


Figure A.7: Passive Monitoring FPX-in-a-Box Configuration

modification, monitored network traffic can be reflected between FPX cards, potentially causing operational problems by consuming available bandwidth.

The NID routing information for this configuration is also shown in Figure A.7. For each of the FPX cards, the NID can route data from each of the SW, LC, RAD_SW, and RAD_LC input interfaces to any of the same output ports. On FPX 0, the NID should be configured to route VCI 51 (0x33) traffic from the LC interface to the RAD_SW interface. Additionally, VCI 50 (0x32) from the RAD_SW interface and VCI 52 (0x34) from the RAD_LC interface should both be routed to the SW interface. Routes should be setup for the FPX 1 and FPX 2 devices as indicated in the diagram.

A.5 Known Problems

Below is a list of known problems with the current distribution. Most of these problems do not reflect errors in the design or the code, but are due to changes in the tools provided by Xilinx and Synplicity which are incompatible with previous versions of these same tools.

A.5.1 Xilinx ISE 6.2i

Xilinx changed the dual-port RAM block produced by CoreGen. The previous dual-port block RAM memory device contained a version 1 implementation. The current dual-port RAM memory device uses version 5 constructs. The `vhdl/wrappers/IPProcessor/vhdl/outbound.vhd` source file directly references the version 1 component which is no longer supported by newer versions of the tools. To circumvent this problem, build an identically-sized version 5 component and add the vhd and edn files to the project. Xilinx also changed the signal names with the new component. This can be resolved by editing the `outbound.vhd` file and fix up the ram512x16 component and its implementations to correspond to the version 5 CoreGen component.

A.5.2 Synplicity 7.5

Synplicity v7.5 is not compatible with the *alias* construct used in the `vhdl/wrappers/FrameProcessor/vhdl/aal5dataen.vhd` source file because previous versions of Synplicity had no problem with this VHDL construct. One workaround is to switch to an older or newer version of Synplicity (version 7.3 or 7.6 for example). Another workaround is to modify the source file to remove the alias construct and directly address the proper signal names.

A.5.3 Outbound IPWrapper Lockup

There have been several occurrences of a situation where the outbound lower layered protocol wrappers stop processing traffic and de-assert the TCA flow control signal to the TCP-Processor circuit. This problem has been observed when monitoring live network traffic with a configuration similar to that shown in Figure A.6 where encoded traffic exiting the Scan circuit is routed back to the StreamExtract circuit for egress processing. After several days of processing traffic, the outbound IP Wrapper will de-assert the flow control signal and lock up the whole circuit. It is not clear whether or not this lockup is due to

invalid data being sent from the TCP-Processor egress component to the IP Wrapper or a bug in the outbound lower layered protocol wrapper code. The exact sequence of events which causes this problem has not been determined. One plausible explanation for this problem is the presence of an IP packet fragment.

A.5.4 Inbound Wrapper Problems

During campus network traffic testing, it was discovered that AAL5 CRC errors are not dealt with in the inbound AAL5 frame wrapper, nor in the inbound IP wrapper. The AAL5 frame wrapper performs the CRC calculation, and updates the CRC word which occurs after the end of the network packet. A valid CRC computation results in this CRC value being replaced with the value of zero. An invalid CRC computation results in a non-zero value. The AAL5 frame wrapper passes this invalid CRC indication at the tail end of the packet.

The IP wrapper does not look at the CRC calculation and assumes that all data it receives from the AAL5 frame wrapper contain valid packets. The processing of an invalid packet can confuse the state machine in the IP wrapper and cause severe problems for downstream client processing logic.

Appendix B

Generating Simulation Input Files

The ability to quickly and easily generate simulation input files is paramount to any debugging effort. This section focuses on capturing network traffic and converting it into a simulation input file used by ModelSim and the testbench logic. Simulation input files can be generated by hand, but this is tedious work. The following subsections contain usage information for the `tcpdump`, `dmp2tbp`, IPTESTBENCH, and `sramdump` applications as well as the CellCapture circuit for generating simulation input files.

B.1 tcpdump

The `tcpdump` [83] utility captures and displays the contents of network packets arriving on a network interface on a host machine. A `tcpdump` command like the one below will capture the network traffic to a file.

```
$ tcpdump -i eth1 -X -s 1514 host 192.168.20.10 > tmp.dmp
```

Change the `tcpdump` parameters as needed to specify the proper host. Specify the **`-X`** flag to indicate a hexadecimal output format and the **`-s 1514`** parameter to capture the complete contents of each packet. Below is sample of network traffic captured with the aforementioned command:

```
16:04:34.151188 192.168.204.100.3474 >
hacienda.pacific.net.au.http: S 1180902188:1180902188(0) win 16384
<mss 1460,nop, nop,sackOK> (DF)

0x0000 4500 0030 e0a1 4000 7d06 52c9 c0a8 cc64 E..0..@.}.R....d
0x0010 3d08 0048 0d92 0050 4663 232c 0000 0000 =..H...PFc#,....
```

```
0x0020 7002 4000 0151 0000 0204 05b4 0101 0402 p.@..Q.........

16:04:34.362576 hacienda.pacific.net.au.http >
192.168.204.100.3474: S 545139513:545139513(0) ack 1180902189 win
5840 <mss 1460,nop,nop,sackOK> (DF)

0x0000 4500 0030 0000 4000 2b06 856b 3d08 0048 E..0..@.+..k=..H
0x0010 c0a8 cc64 0050 0d92 207e 2b39 4663 232d ...d.P...~+9Fc#-
0x0020 7012 16d0 deb8 0000 0204 05b4 0101 0402 p.............

16:04:34.364557 192.168.204.100.3474 >
hacienda.pacific.net.au.http: . ack 1 win 17520 (DF)

0x0000 4500 0028 e0a3 4000 7e06 51cf c0a8 cc64 E..(..@.~.Q....d
0x0010 3d08 0048 0d92 0050 4663 232d 207e 2b3a =..H...PFc#-.~+:
0x0020 5010 4470 dddc 0000 2020 2020 2020      P.Dp.........
```

B.2 dmp2tbp

The `dmp2tbp` application has been developed to convert the `tcpdump` output format into a simulation input file for use by ModelSim. A `dmp2tbp` command takes two parameters, an input file which was generated by a `tcpdump` command and a target output file to contain the IPTESTBENCH input data. The syntax of the application is shown below:

```
Usage:  dmp2tbp <input dmp file> <output tbp file>
```

Links to the executable and source for the `dmp2tbp` application can be found at: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/dmp2tbp.html`.

The converted output of the previous `tcpdump` sample trace is shown below. For each IP packet, a two-word LLC header was prepended to the packet to match the packet format of the switch environment. In order to account for these two header words prior the start of IP header, the **NUM_PRE_HDR_WRDS** configuration parameter will need to be set to a value of 2. The LLC header information can easily be removed by editing the `dmp2tbp` source file.

```
!FRAME 50
AAAA0300
```

```
00000800
45000030
e0a14000
7d0652c9
c0a8cc64
3d080048
0d920050
4663232c
00000000
70024000
01510000
020405b4
01010402
!FRAME 50
AAAA0300
00000800
45000030
00004000
2b06856b
3d080048
c0a8cc64
00500d92
207e2b39
4663232d
701216d0
deb80000
020405b4
01010402
!FRAME 50
AAAA0300
00000800
45000028
e0a34000
7e0651cf
c0a8cc64
3d080048
0d920050
4663232d
207e2b3a
50104470
dddc0000
20202020
```

```
20200000
```

B.3 IPTESTBENCH

The `ip2fake` executable of the IPTESTBENCH [14] utility converts the output of the `dmp2tbp` application into the simulation input file used by ModelSim. Rename the file generated by the `dmp2tbp` program to INPUT_CELLS.TBP and copy the file to the `sim` directory. Run the following commands to convert the .TBP file to the ModelSim .DAT input file:

```
$ cd sim
$ make input_cell
```

A file called INPUT_CELLS.DAT is generated which contains properly formatted packets that can be used to perform simulations. The source code and make files for IPTESTBENCH are included with each circuit design in the `iptestbench` subdirectory.

B.4 sramdump

The `sramdump` utility reads data out of SRAM and writes the results to a disk file. To accomplish this task, the `sramdump` application communicates with NCHARGE over a TCP socket connection. NCHARGE in turn communicates with the Control Cell Processor (CCP) circuit programmed into a FPX card. This communication utilizes ATM control cells to read the SRAM memory of the FPX card. Results are passed to NCHARGE and then the `sramdump` application via the return path. Figure B.1 shows a diagram of this communication path. The `sramdump` utility requires that the CCP circuit be built into the RAD application and that the CCP control cells (typically VCI 0x23) be correctly routed through the switch, the NID and the RAD. Links to the executable and source for the `sramdump` application can be found at: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/sramdump.html`.

The `sramdump` program automatically detects the endianess of the host machine and formats output so that the resulting dump file is consistent, regardless which machine on which the utility was run . There are various dumping options, a binary dump [useful as input to other analysis programs], an ASCII dump [good for text data], a dump format

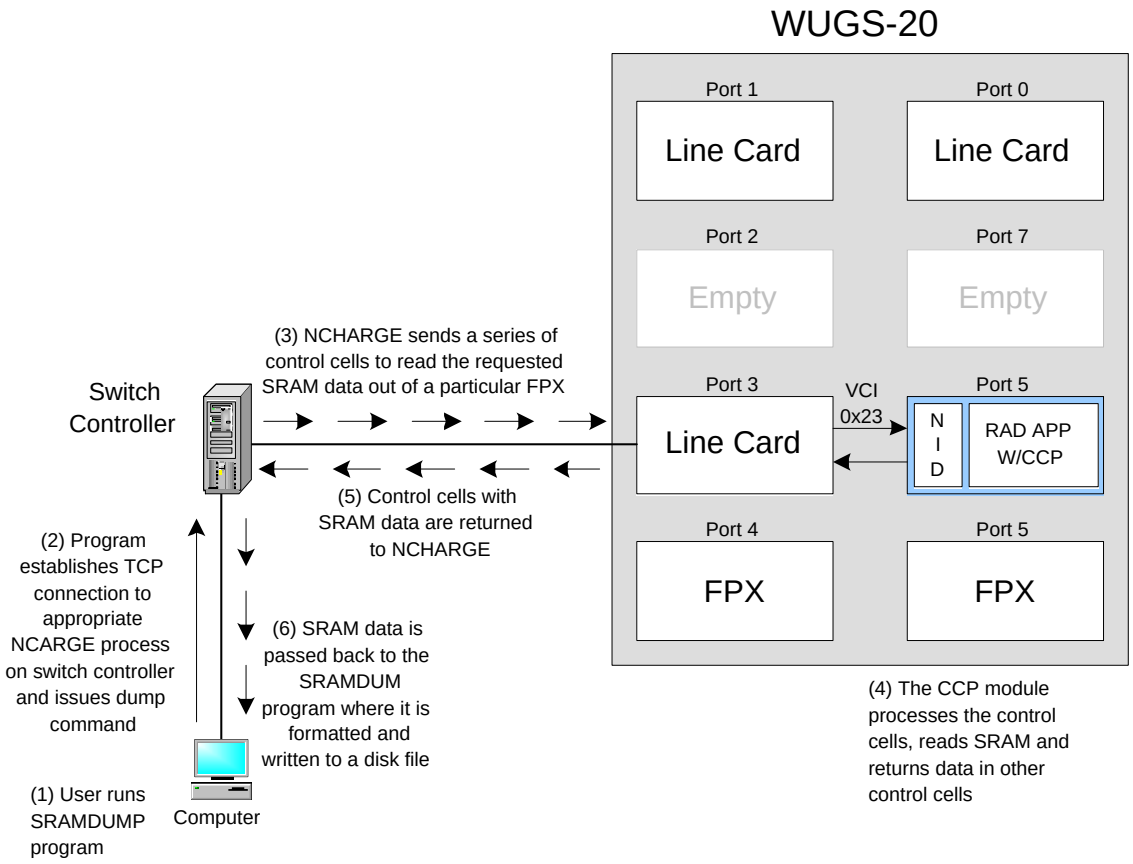


Figure B.1: Operation of SRAMDUMP Utility

[shows both hexidecimal and ASCII representations of data], and an option to generate IPTESTBENCH input files.

SRAM memory on the FPX platform is 36 bits wide. Depending on the applicaitons's use of this memory, data can either be read out in 32-bit segments (ignoring the upper 4 bits) or in 36-bit segments. When reading 36-bit data, the high nibble is normally stored as a byte-sized value. The **-e** option will expand the output and represent this as a 4-byte quantity such that the low bit of each byte will represent the value of one of four high bits. This formatting option makes it easier to interpret a series of 36-bit data values.

In order to generate data formatted as AAL5 frames for input into IPTESTBENCH, data must be written in a special SRAM frame format. Likewise, in order to generate data formatted as ATM cells for input into ModelSim, data must be written in a special SRAM cell format. Figure B.2 shows the specific SRAM frame and cell formats. Output files

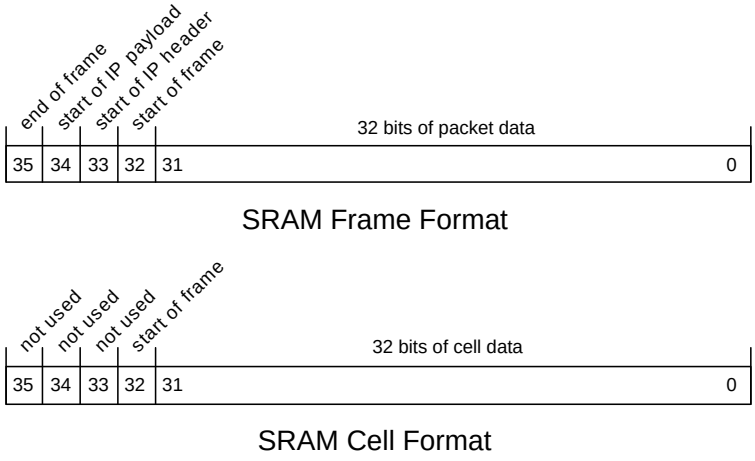


Figure B.2: SRAM data Formats

generated using the **-t F** or **-t C** options can be used as the input file to IPTESTBENCH for creating simulation input files for ModelSim.

Usage information for the `sramdump` application is shown below:

```
Usage: sramdump [OPTIONS]
    -h <string> hostname of switch controller:
                                    [illiac.arl.wustl.edu]
    -c <num> card number (0 - 7): [0]
    -b <num> memory bank (0 - 1): [0]
    -a <num> address to start reading (in hex): [0]
    -n <num> number of words to read: [1]
    -o <string> output file: [sram.dat]
    -t <char> output type (B-binary, A-ascii, D-dump,
                           F-frames, C-cells): [B]
    -f <char> dump format (B-big endian, L-little endian): [B]
    -w <num> dump column width: 4
    -s <num> dump separations (spaces every n bytes): [4]
    -l <num> length of word (32, 36): [32]
    -e expanded 36 bit format
       (high 4 bits expanded to low bit of 4 byte field)
    -? this screen
```

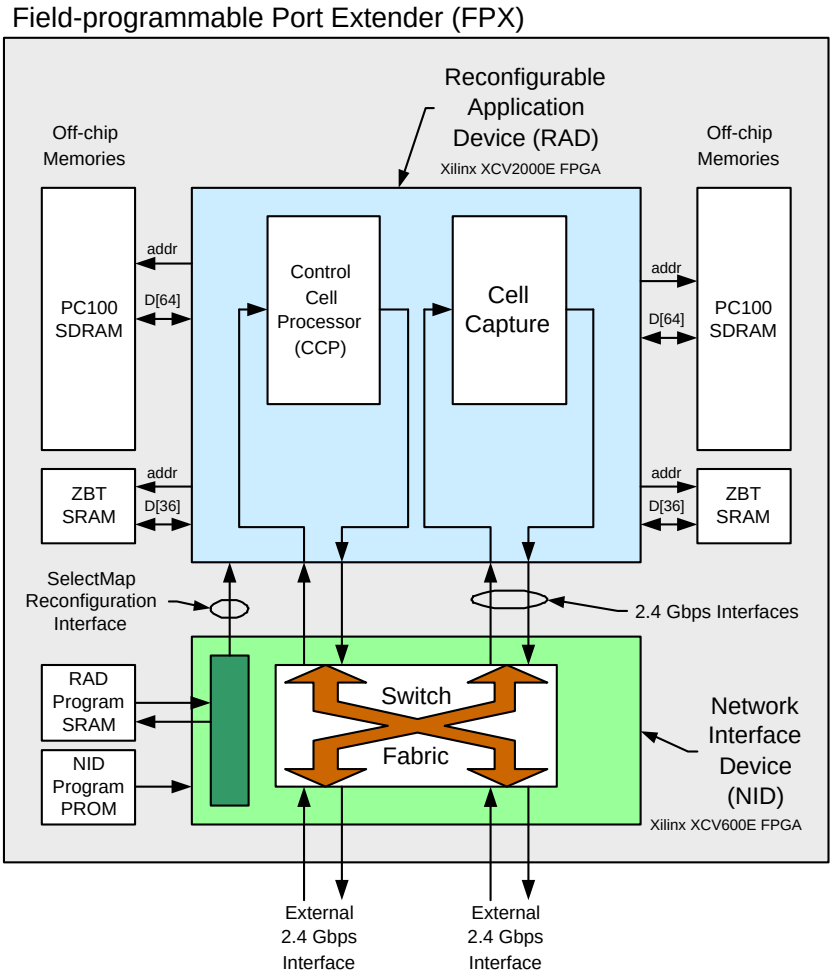


Figure B.3: CellCapture Circuit Layout

B.5 Cell Capture

The CellCapture circuit provides an easy mechanism for capturing traffic with an FPX card. The circuit writes captured ATM cells to the SRAM0 memory device and then to the SRAM1 memory device. Each SRAM unit contains 256 thousand 36-bit memory locations which can store 18,724.5 cells. The CellCapture circuit will store the first 37,449 cells that are passed into the RAD_LC interface. Because the SRAM devices support memory operations on every clock cycle, the circuit is capable of capturing traffic at full line rates (writing data to memory on every clock cycle). The circuit synthesizes at 100MHz which supports the fastest traffic rates possible on the FPX platform. Figure B.3 shows a layout of the CellCapture circuit.

The CellCapture circuit is structured in a similar manner to all other circuits described in this document. Links to the project source for the CellCapture circuit can be found at: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/cell_capture.html`. The steps required to build an executable *bit* file from the source are listed below:

```
$ make compile
$ make syn
$ make build
```

To utilize this circuit, route control cell traffic (VCI 0x23) into the RAD_SW port and route traffic to be captured (the VCI(s) of interest) into the RAD_LC port. The RAD-controlled LEDs 1 and 2 blink continuously to indicate that the circuit is operating. The RAD-controlled LEDs 3 and 4 illuminate when SRAM bank 0 and 1 respectively, are full of data. LED 3 will always illuminate before LED 4 because the circuit fills up SRAM bank 0 before writing to SRAM bank 1.

After capturing network traffic, the `sramdump` utility can be used to retrieve the captured data. The resulting data file can be converted into a ModelSim input file utilizing IPTESTBENCH. Attempts to read data from a SRAM before it is full of data will cause the CCP read operations to SRAM take priority over the CellCapture writes. This implies that inconsistent network captures may be produced if a SRAM read operation occurs at the same time as a SRAM write operation. This circuit is designed to work with the `sramdump` utility. Use the **-l 36 -t C -e** options with the `sramdump` command to generate ModelSim input files.

B.6 Internal Data Captures

The internal data captures supported by many of the TCP-Processor circuits provide an excellent method for gaining insight into the internal operation of the circuits. These data captures can be extracted from the FPX devices utilizing the `sramdump` routine and converted into simulation input files utilizing IPTESTBENCH. The TCPInbuf, TCPEgress, TCPSerializeEncode, TCPSerializeDecode, TCPDeserialize, and TCPReserialize components all support the storing of network traffic to SRAM memory devices. The TCPInbuf and the TCPEgress components utilize the SRAM frame format for storing network traffic. The TCPSerializeEncode, TCPSerializeDecode, TCPDeserialize, and TCPReserialize components utilize the SRAM cell format for storing network traffic.

Appendix C

Statistics Collection and Charting

This appendix contains information on software applications which collect, chart, and redistribute statistics information generated by the various circuits mentioned in this document. These applications directly process the statistics packets generated by the hardware circuits. The statistics packets are transmitted as UDP datagrams, with the packet payload format shown in Table C.1.

Table C.1: Statistics Format

31 24	23 16	15 8	7 0
Statistics Type	Statistics Identifier	Packet Number	Number of Entries
32-bit statistics value 1			
32-bit statistics value 2			
32-bit statistics value 3			
...			

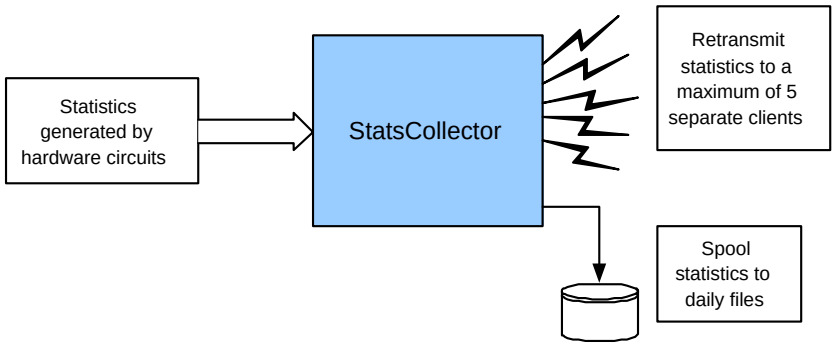


Figure C.1: StatsCollector Operational Summary

C.1 StatsCollector

The StatsCollector is a C program which gathers statistics information and spools the data to disk files. A new disk file is created every night so that the data sets can be easily managed. The StatsCollector can also be configured to rebroadcast statistics packets to a maximum of five separate destinations. This aids in the dissemination of statistical information to a wide audience. Figure C.1 shows an operational summary of the StatsCollector.

The source for the StatsCollector application can be found at the following web site: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/statscollector.html`. Usage information for the StatsCollector application is shown below:

```
Usage: statscollector [OPTION]
   -p <num>           UDP port on which to listen for
                      statistics information
   -r <string>:<num>  remote IP host & UDP port on which
                      to re-transmit stats (up to 5 remote
                      hosts can be specified)
   -s                 silent - do not capture statistics to
                      disk file
   -?                 this screen
```

The statistics files generated by the StatsCollector application have a specific filename format shown below:

`stats%%&&_YYYYMMDD`

%% represents a one-byte (two-hex-digit)
number indicating the statistics type

&& represents a one-byte (two-hex-digit)
number indicating the statistics identifier
YYYY is a 4-digit year
MM is a 2-digit month
DD is a 2-digit day

These files contain ASCII columns of numbers where each column represents a separate statistics value and each row corresponds to a statistics update packet. This format can be directly imported by programs such as Microsoft Excel and can be graphed utilizing gnuplot. A sample perl script is shown below which generates a separate graph of active flows for each daily data file in the current directory:

```
#!/usr/bin/perl

# list of stats files
my @file_list = <stats01*>;

# for each file in the directory
foreach $file (@file_list) {

# extract info from filename
  my $type = substr($file, 5, 2);
  my $id = substr($file, 7, 2);
  my $year = substr($file, 10, 4);
  my $mon = substr($file, 14, 2);
  my $day = substr($file, 16, 2);

# generate gnuplot control file to create chart
  open handle, "> tmp";
  select handle;
  print "set title \"Traffic Stats for $mon/$day/$year\"\n";
  print "set output \"total_flows_$year$mon$day.eps\"\n";
  print "set data style lines\n";
  print "set terminal postscript eps\n";
  print "set xlabel \"Time of day\"\n";
  print "set ylabel \"Total Active Flows\"\n";
  print "set key left\n";
  print "set xdata time\n";
  print "set timefmt \"%H:%M:%S\"\n";
  print "set xrange [\"00:00:00\":\"23:59:59\"]\n";
  print "set format x \"%H:%M\"\n";
```

```
  print "\n";
  print "plot \"$file\" using 1:13 title "Active Flows\n";
  close handle;

# execute gnuplot command to generate chart
  system "gnuplot tmp";
# end loop
}
```

C.2 StatApp

The StatApp is a Java application created by Mike Attig (mea1@arl.wustl.edu) to perform real-time graphing. The original StatApp program was modified to accept data generated in the standard statistics format and chart the data in real-time. The following commands can be used to compile and run the StatApp charting application. The port number should be set to the UDP port number of the statistics packets, currently 6501.

```
$ javac *.java
$ java StatApp <port#>
```

Figure C.2 displays the output of the StatApp charting various statistics counters from the TCP-Processor. A maximum of 10 data items can be charted in parallel. The various signals are annotated in this diagram for easier reading. The source for the StatApp application can be found at the following web site: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/statapp.html`.

C.3 SNMP Support

A Simple Network Management Protocol (SNMP) agent aids in the dissemination of the statistical information gathered in the various hardware circuits. This agent listens for the statistics packets generated by the various hardware circuits, maintains internal counters for these statistics, and re-exposes this data as SNMP Management Information Base (MIB) variables. This agent was designed to interface with net-snmp v5.1.1 (found at `http://net-snmp.sourceforge.net/`).

The distribution for the SNMP agent along with the MIB declaration can be found at the following URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal/tcp/source/gv_snmp_agent_v2.zip`.

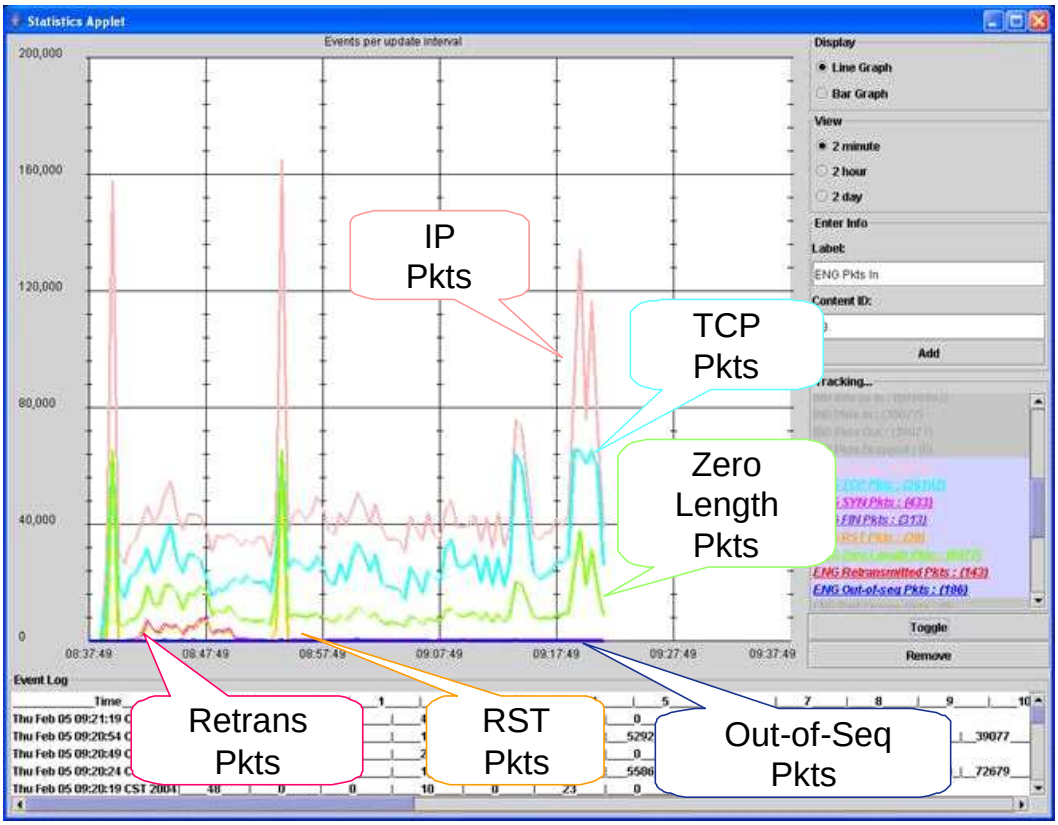


Figure C.2: Sample StatApp Chart

C.4 MRTG Charting

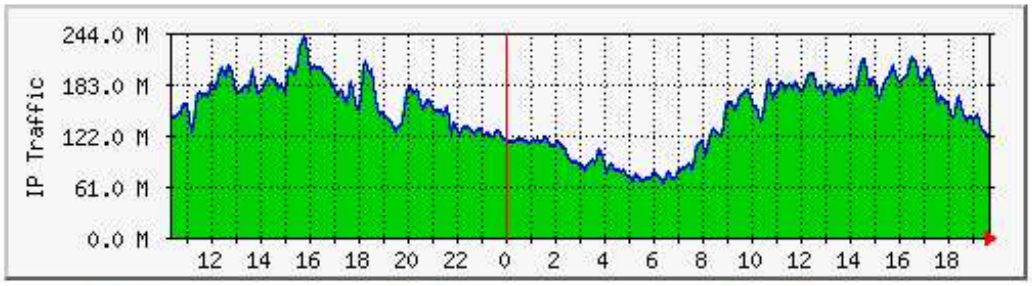
The Multi Router Traffic Grapher (MRTG) tool monitors traffic loads on network links. MRTG generates Hyper Text Markup Language (HTML) web pages on a periodic basis by polling pre-configured SNMP MIB variables. The Portable Network Graphics (PNG) images are generated provide a visual representation of the traffic loads. This tool has been used to generate graphs of the statistical information maintained by the previously mentioned SNMP agent. A sample chart is shown in Figure C.3.

The MRTG charting of statistical information generated by the various hardware circuits was accomplished utilizing the following versions of MRTG and dependent libraries: MRTG v2.10.13, libpng v1.2.5, gd v1.0.22, and zlib v1.1.4. Additional information regarding MRTG can be found at the following web site: `http://www.mrtg.org/`. The MRTG configuration file utilized to chart the statistics information is located at the following URL: `http://www.arl.wustl.edu/projects/fpx/fpx_internal /tcp/source/mrtg_v2.zip`.

IP Traffic

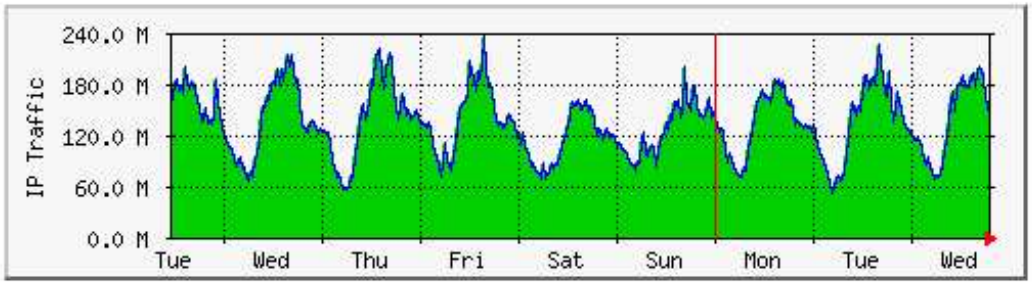
The statistics were last updated **Wednesday, 22 September 2004 at 19:42**

`Daily' Graph (5 Minute Average)



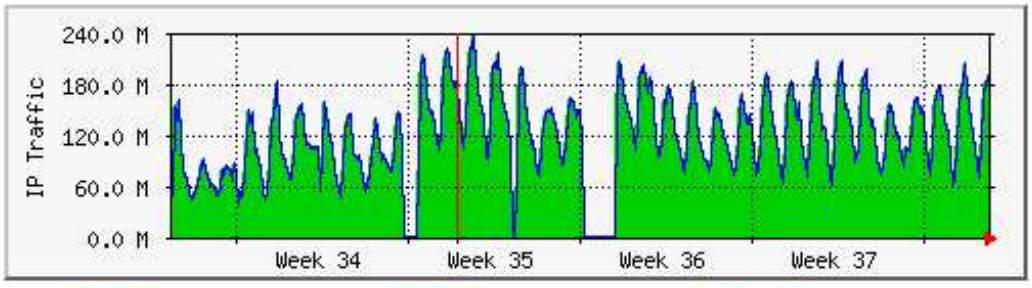
Max IP Traffic: 240.6 Mb/s Average IP Traffic: 150.7 Mb/s Current IP Traffic: 122.4 Mb/s

`Weekly' Graph (30 Minute Average)



Max IP Traffic: 236.2 Mb/s Average IP Traffic: 135.9 Mb/s Current IP Traffic: 144.1 Mb/s

`Monthly' Graph (2 Hour Average)



Max IP Traffic: 238.2 Mb/s Average IP Traffic: 126.5 Mb/s Current IP Traffic: 193.0 Mb/s

Figure C.3: Sample MRTG Generated Chart

Appendix D

Additional Traffic Charts

D.1 Statistics for Aug 27, 2004

D.1.1 Flow Statistics

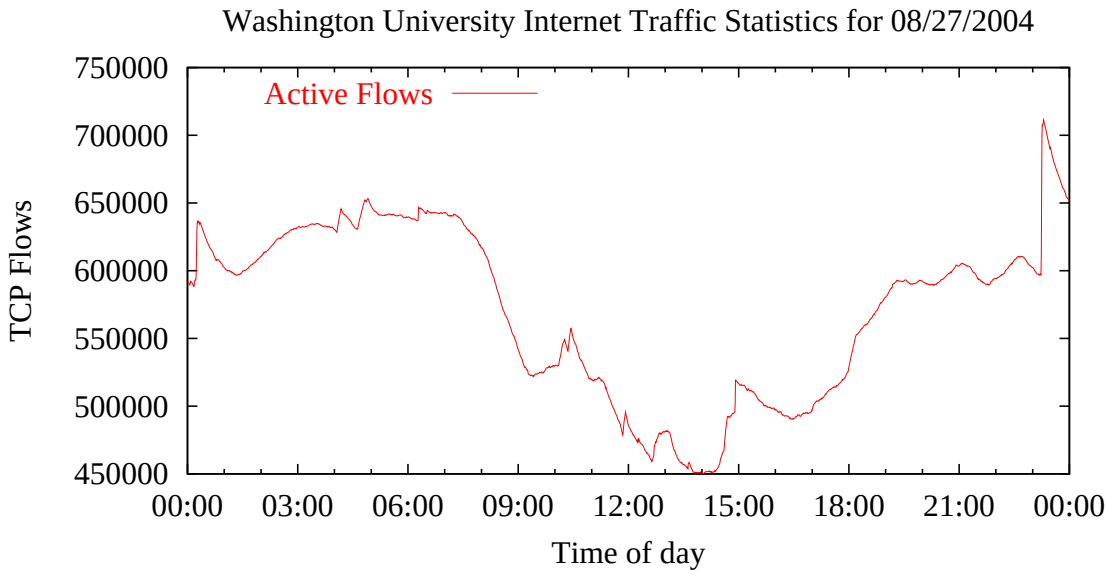


Figure D.1: Active Flows

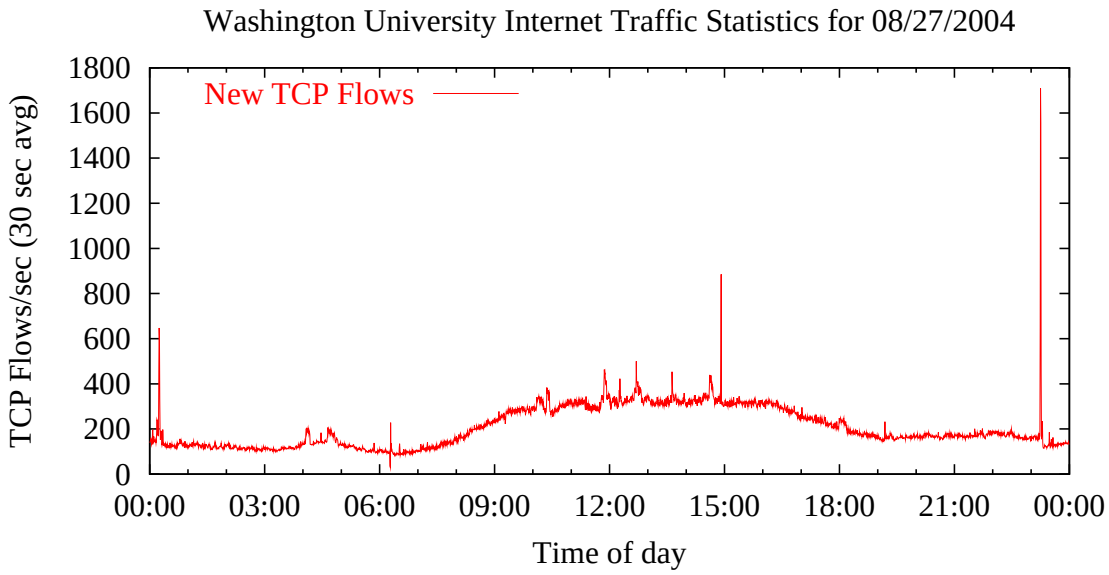


Figure D.2: New Flows

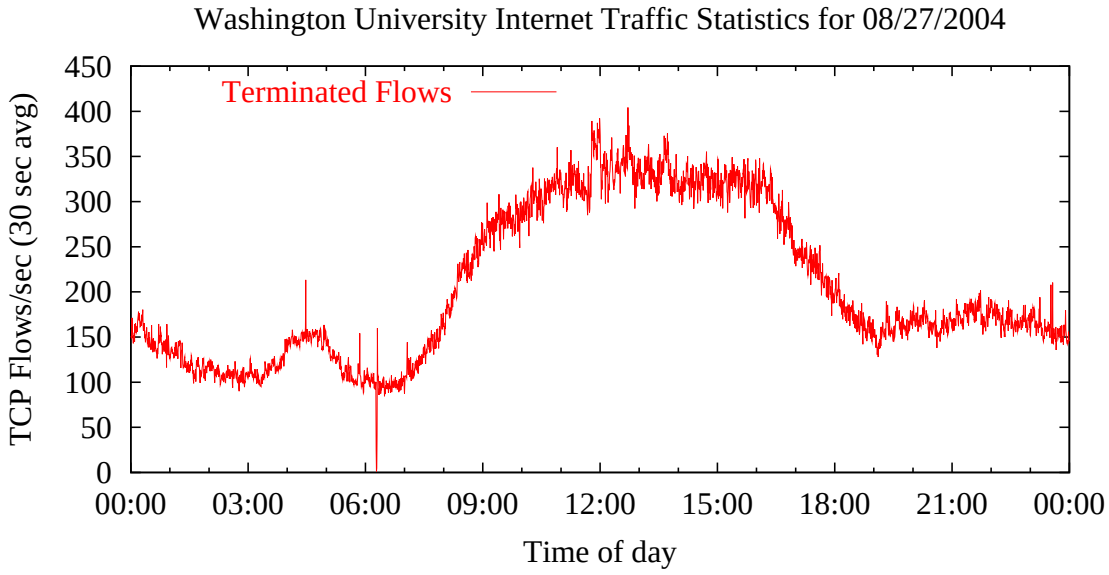


Figure D.3: Terminated Flows

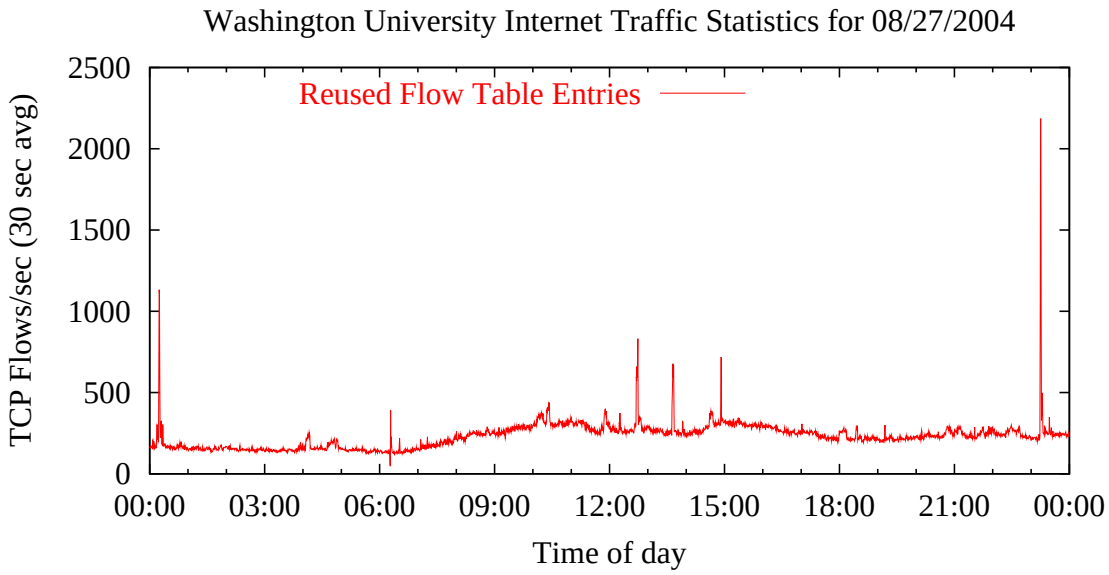


Figure D.4: Reused Flows

D.1.2 Traffic Statistics

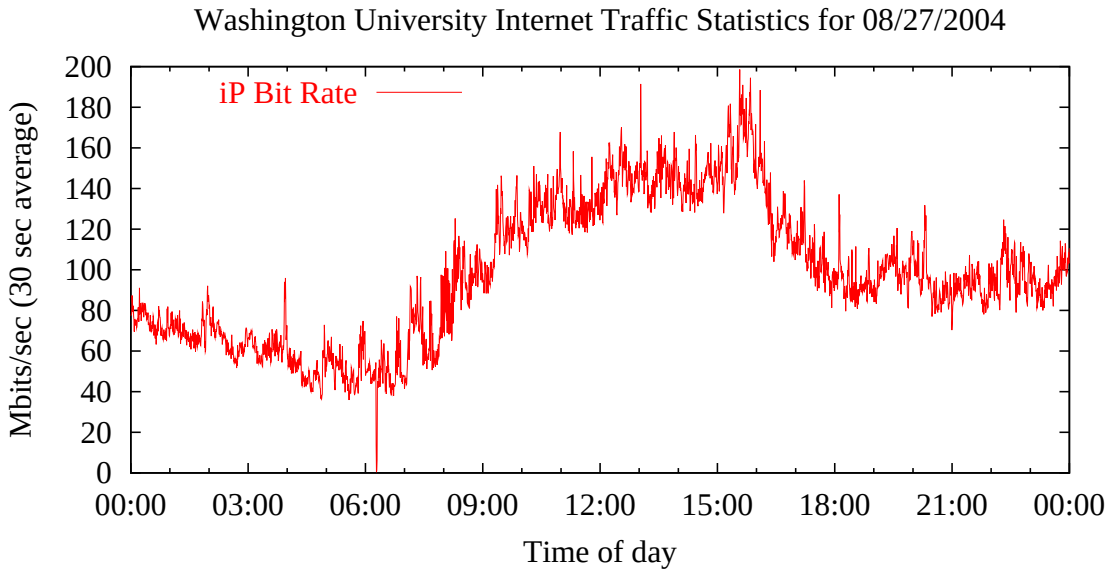


Figure D.5: IP bit rate

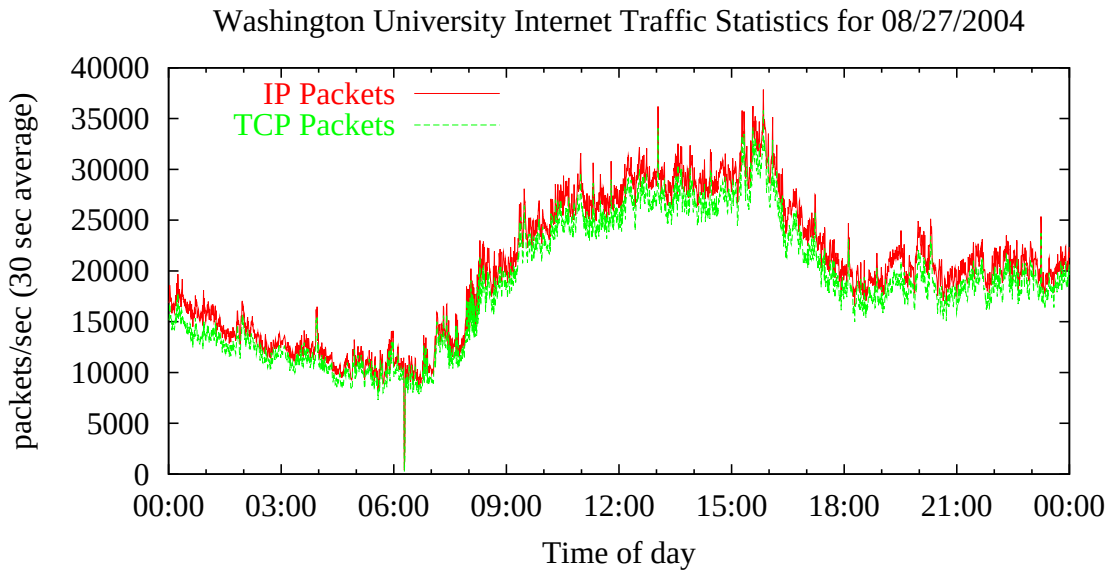


Figure D.6: IP Packets

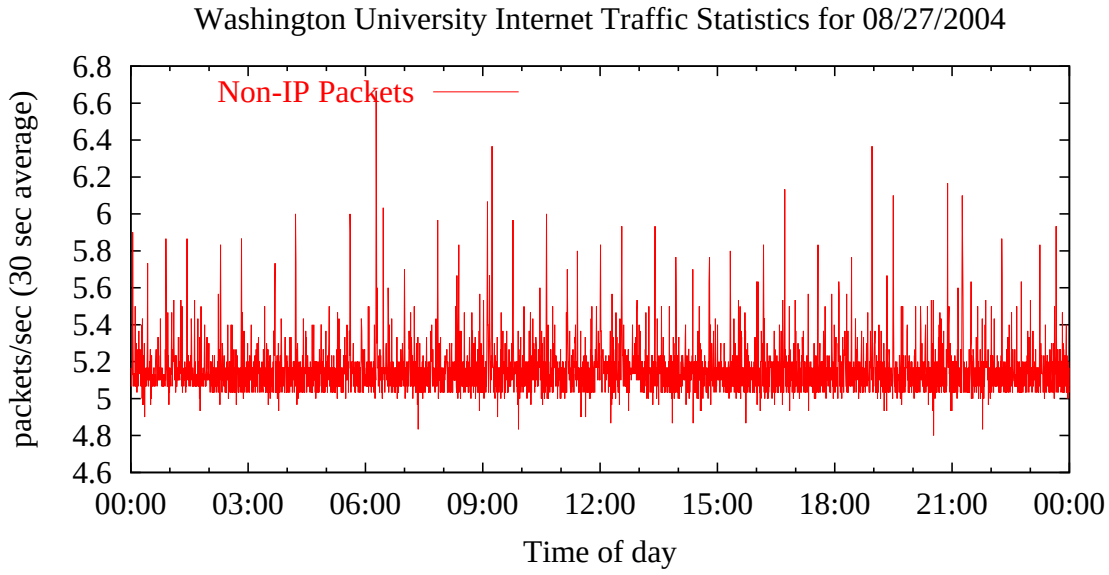


Figure D.7: Non-IP Packets

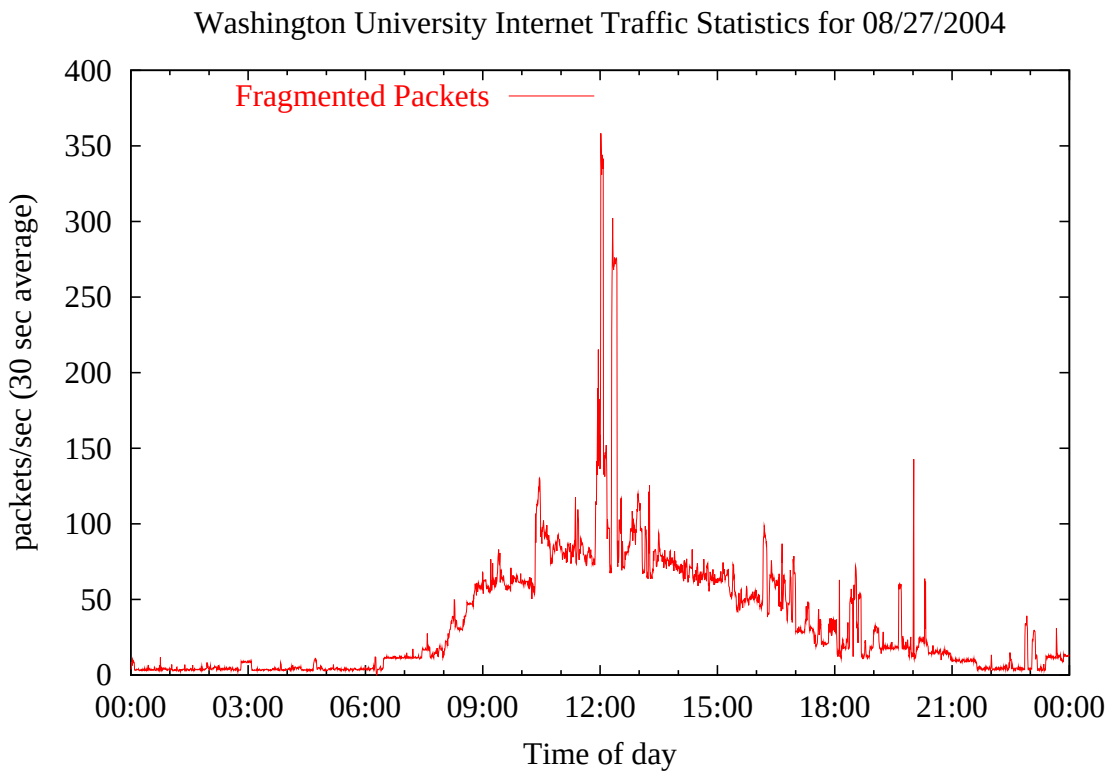


Figure D.8: Fragmented IP Packets

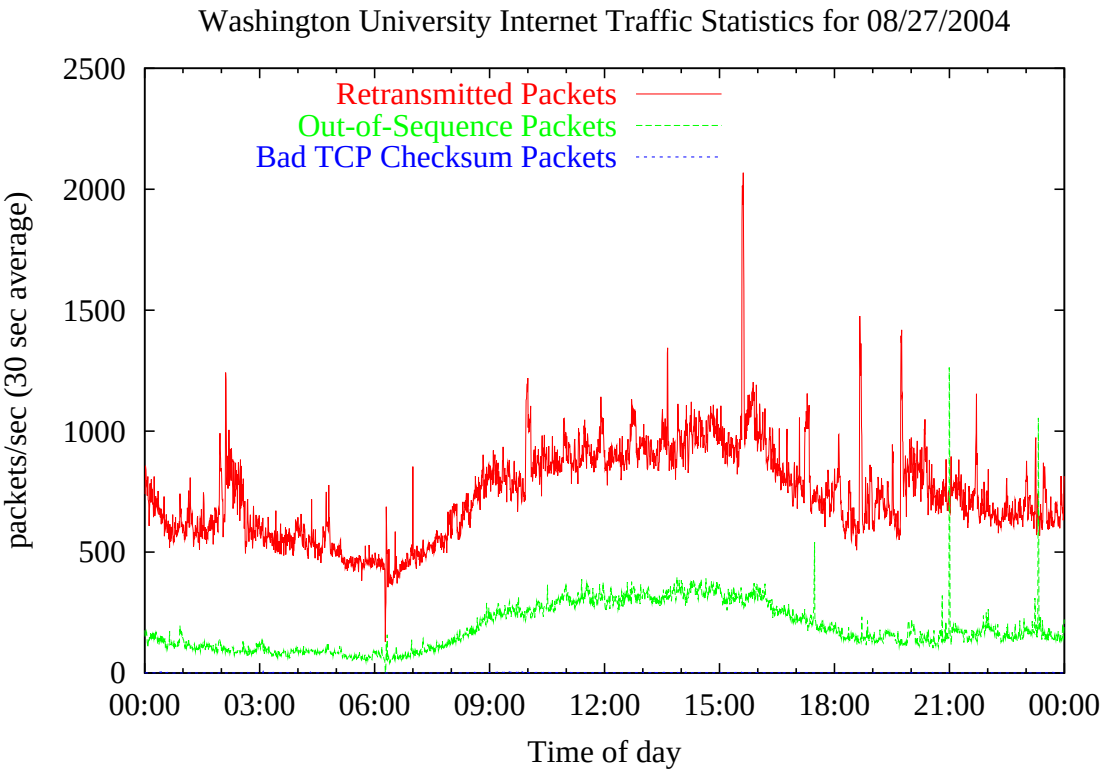


Figure D.9: Bad TCP Packets

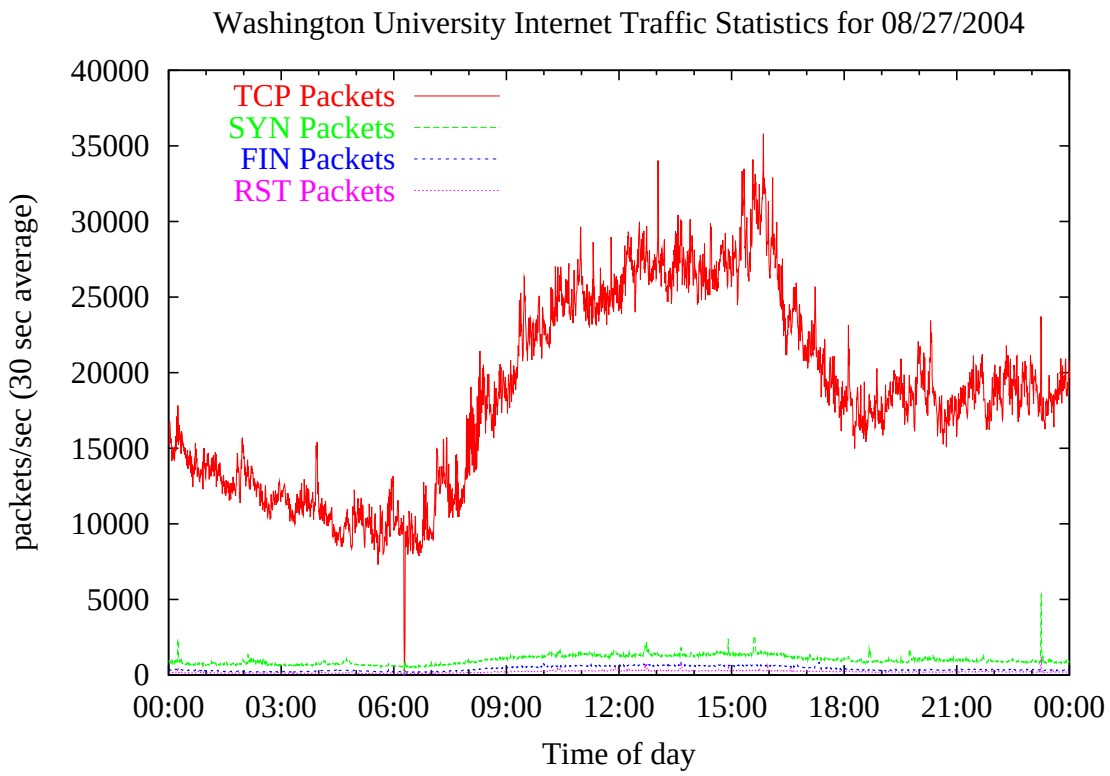


Figure D.10: TCP Packets

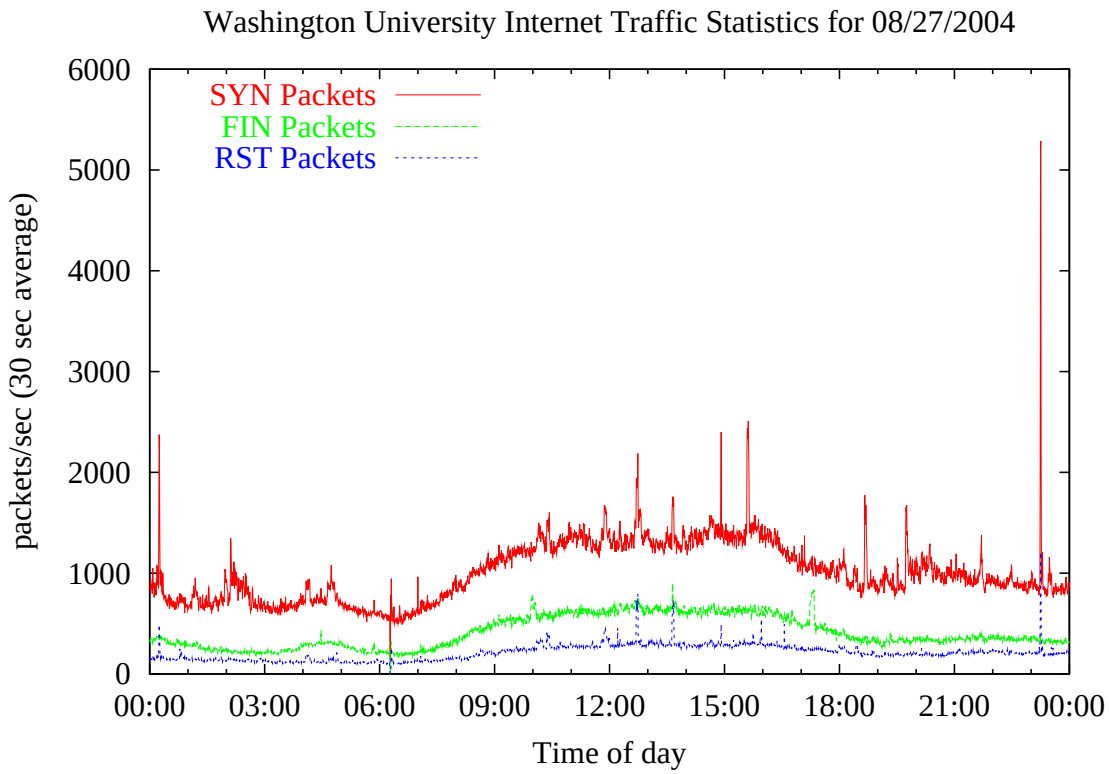


Figure D.11: TCP Packet Flags

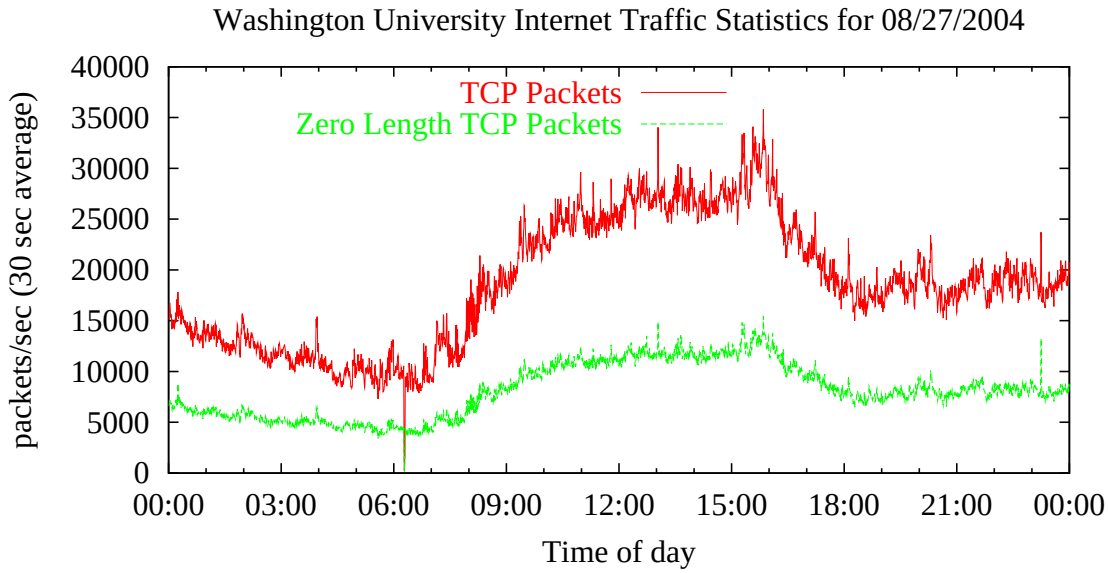


Figure D.12: Zero Length Packets

D.1.3 Port Statistics

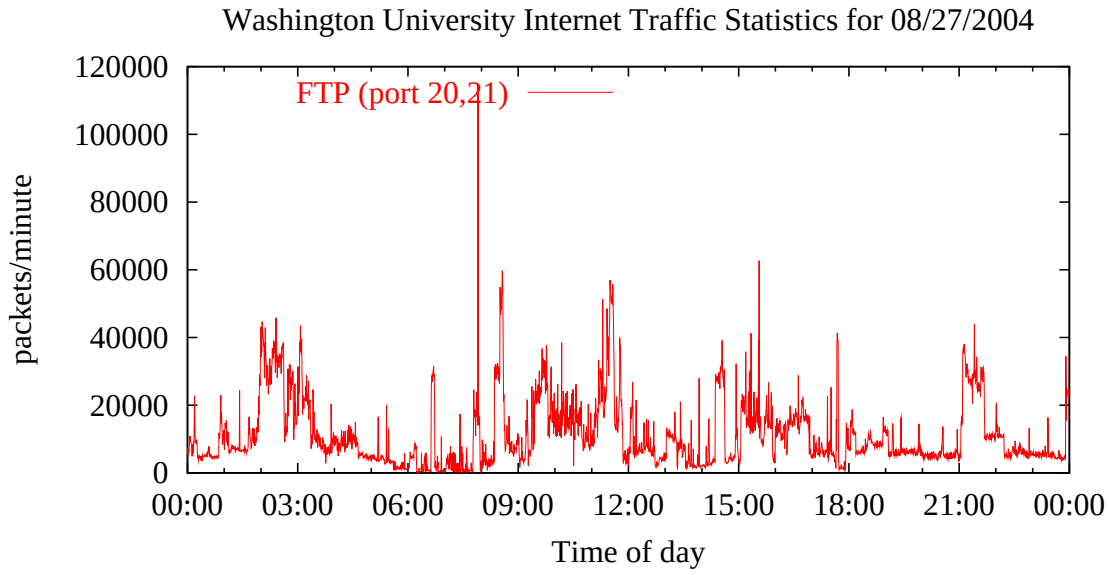


Figure D.13: FTP traffic

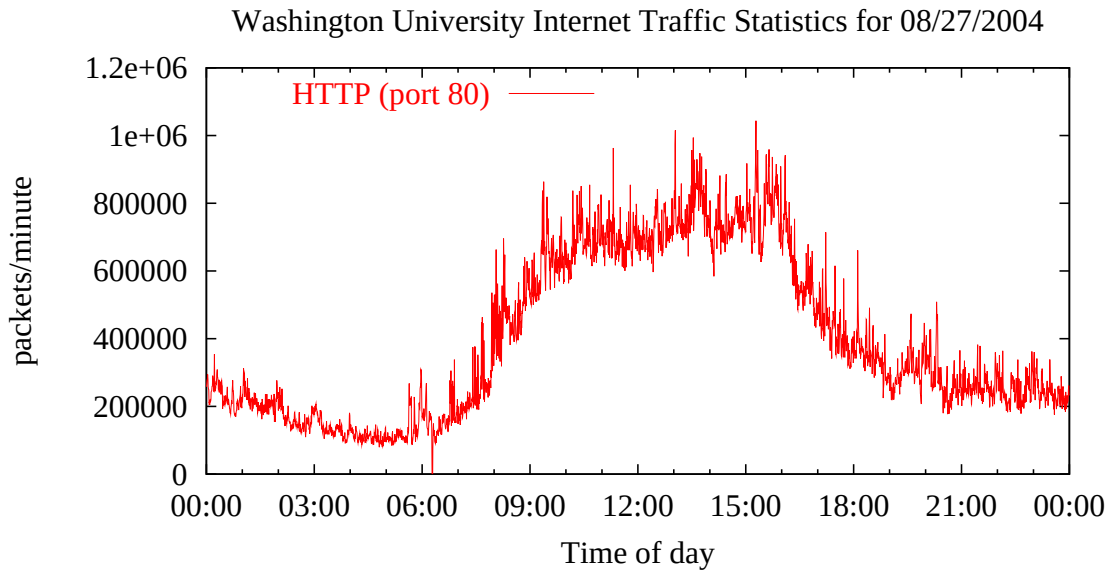


Figure D.14: HTTP traffic

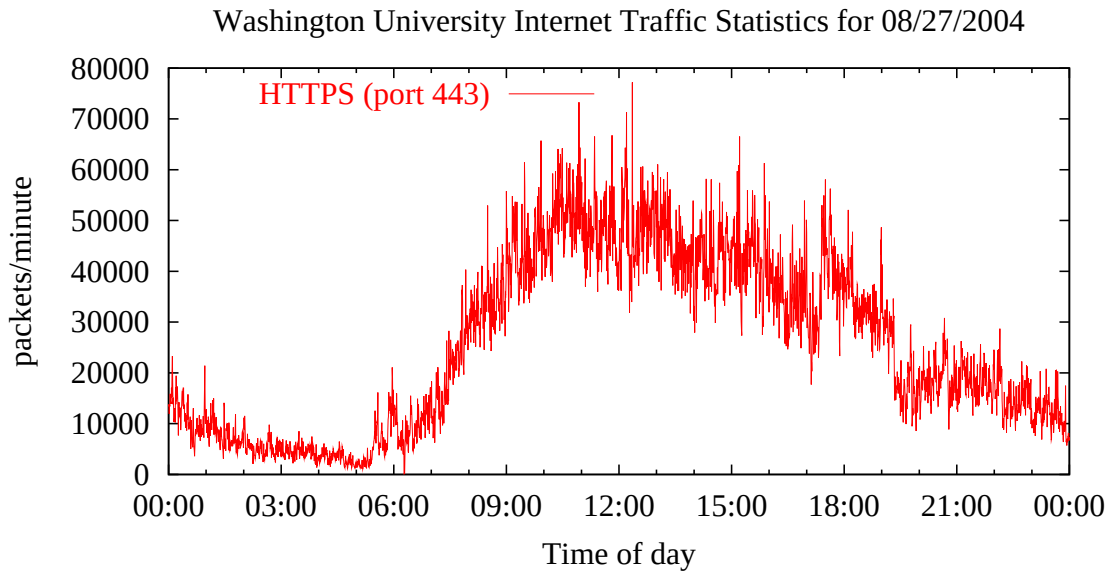


Figure D.15: HTTP traffic

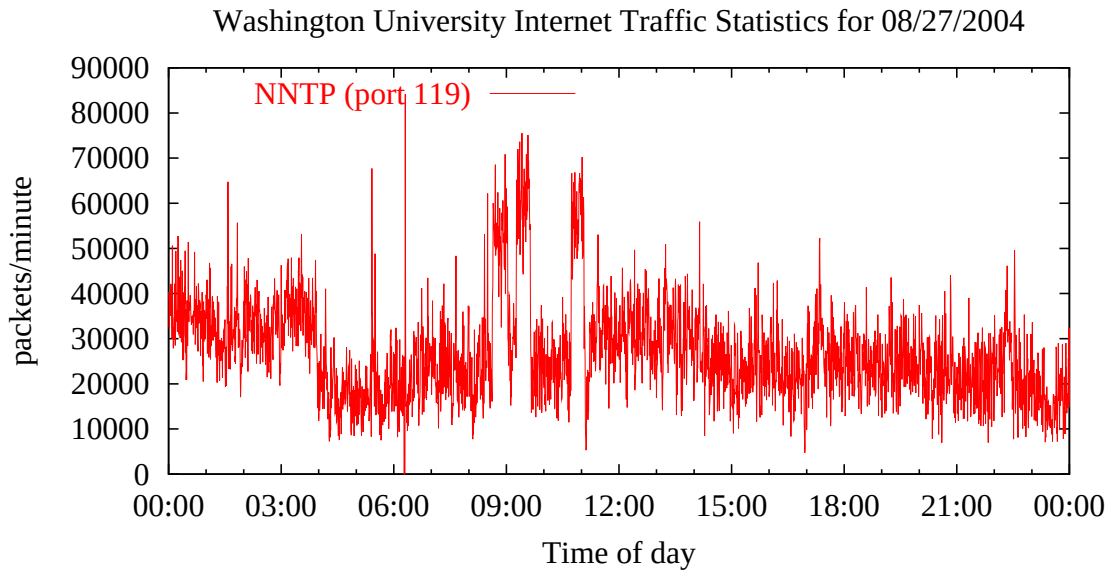


Figure D.16: NNTP traffic

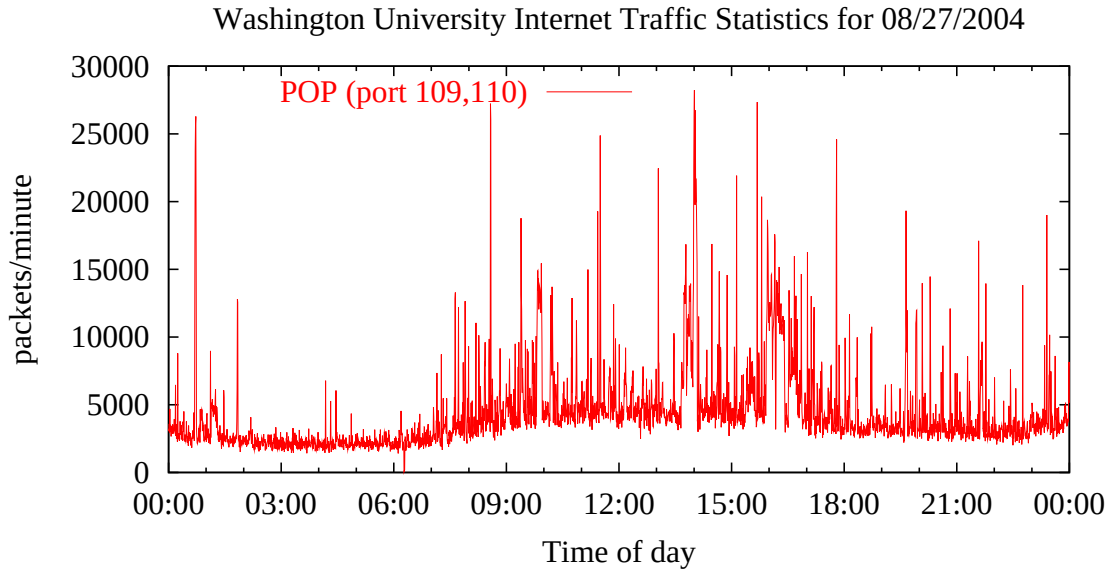


Figure D.17: POP traffic

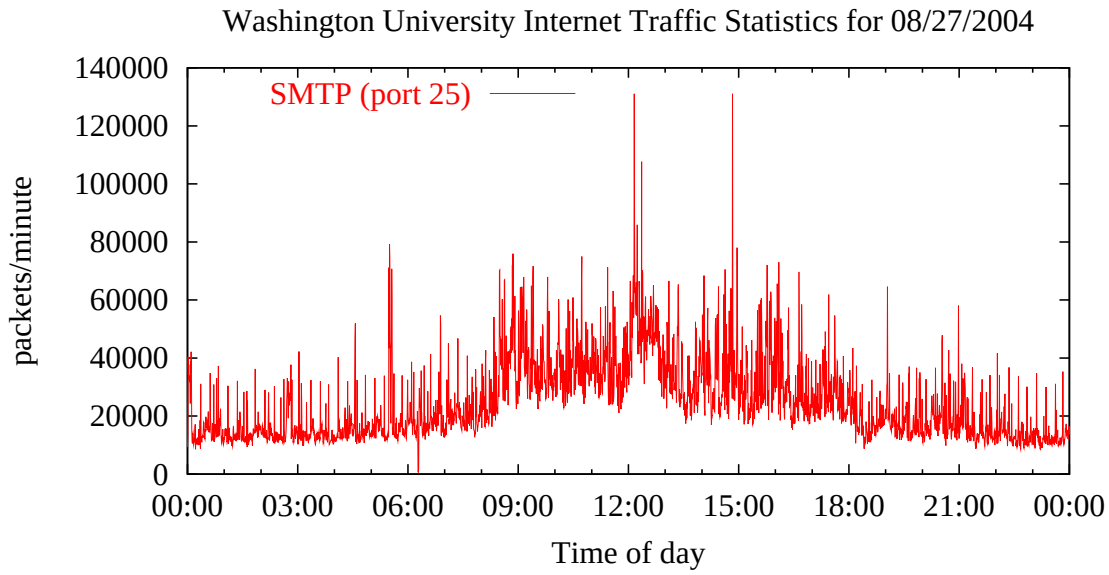


Figure D.18: SMTP traffic

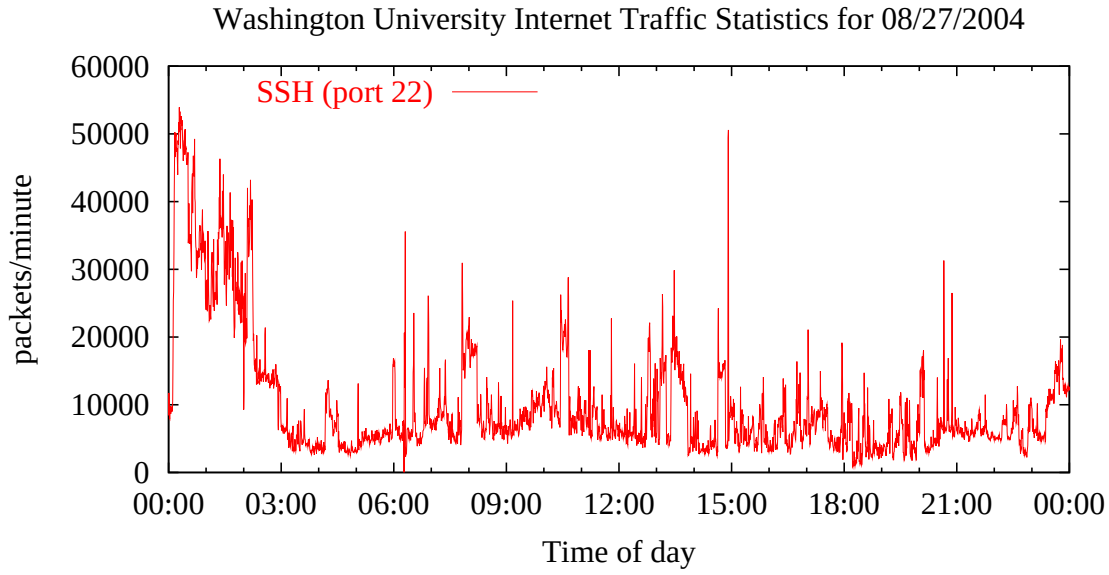


Figure D.19: SSH traffic

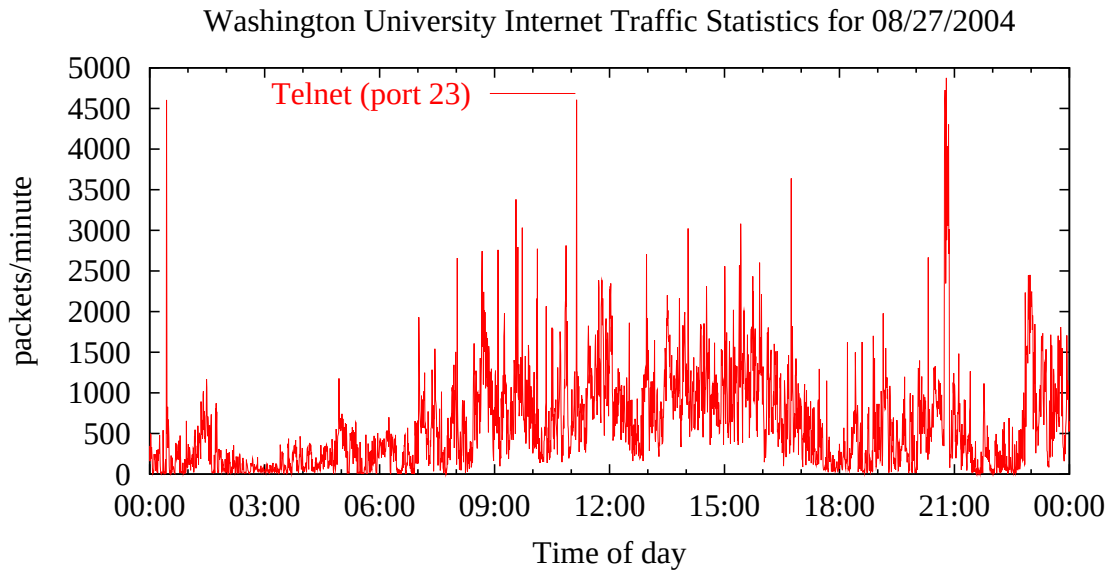


Figure D.20: Telnet traffic

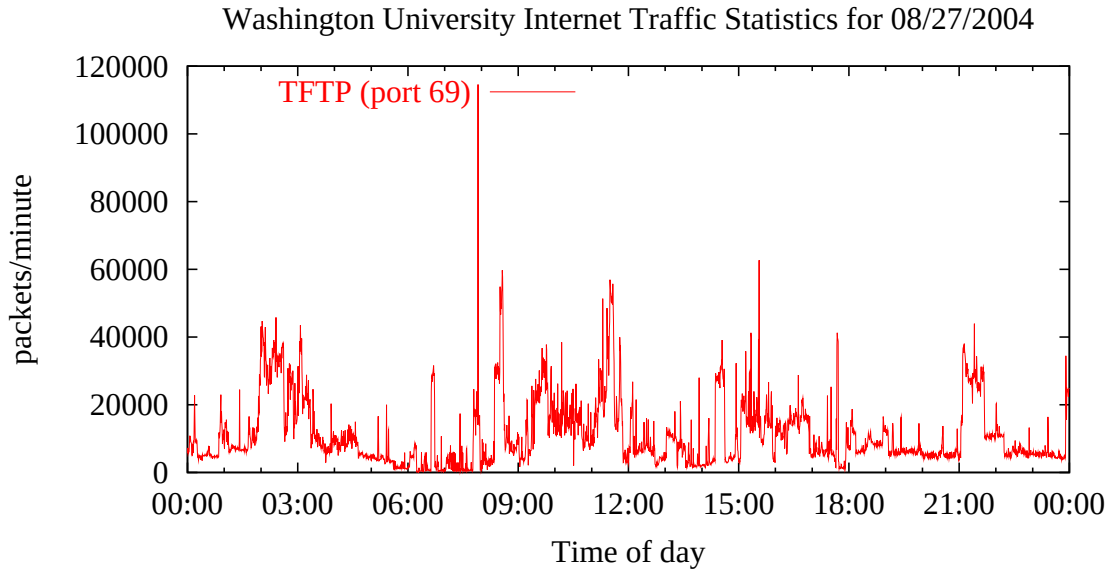


Figure D.21: TFTP traffic

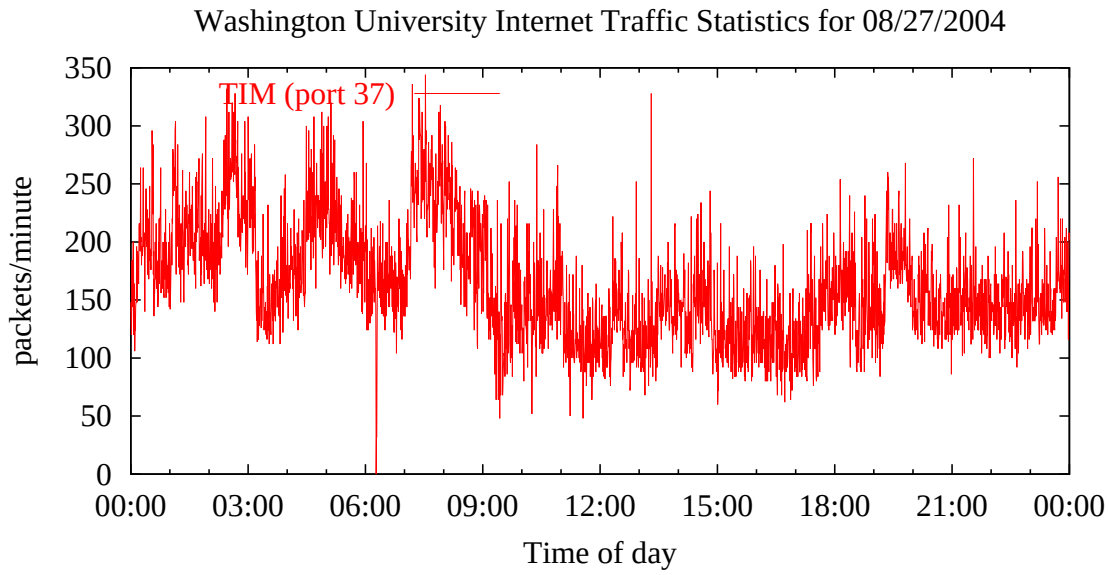


Figure D.22: TIM traffic

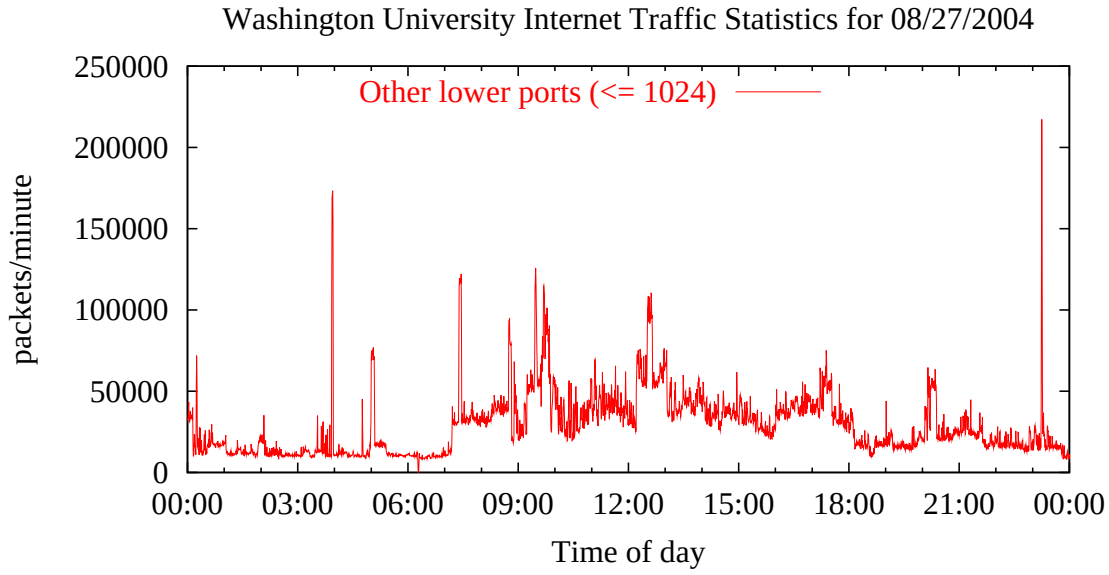


Figure D.23: Lower port traffic

D.1.4 Virus Statistics

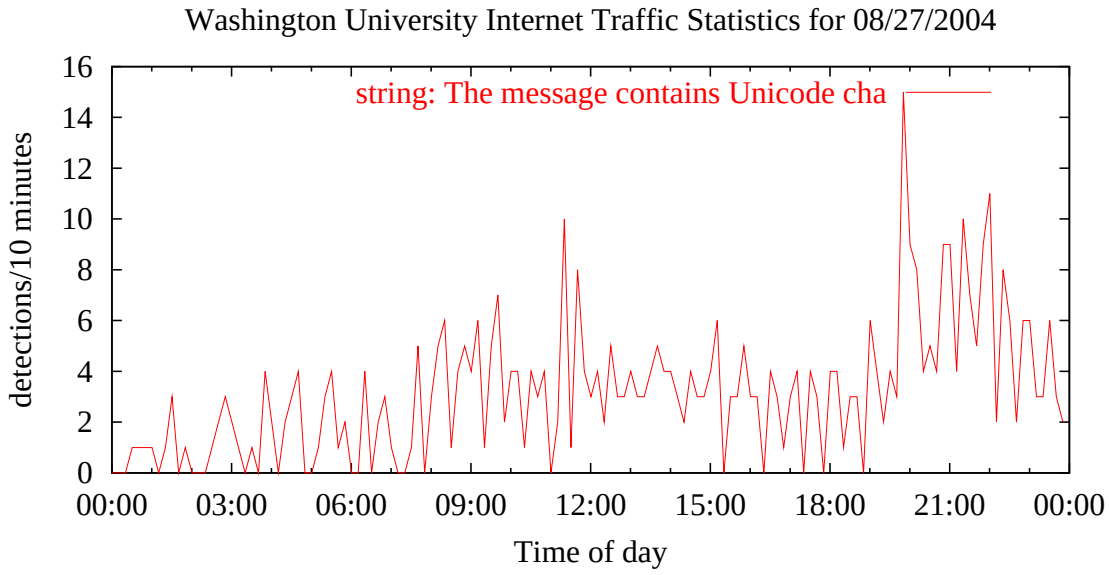


Figure D.24: MyDoom Virus 1

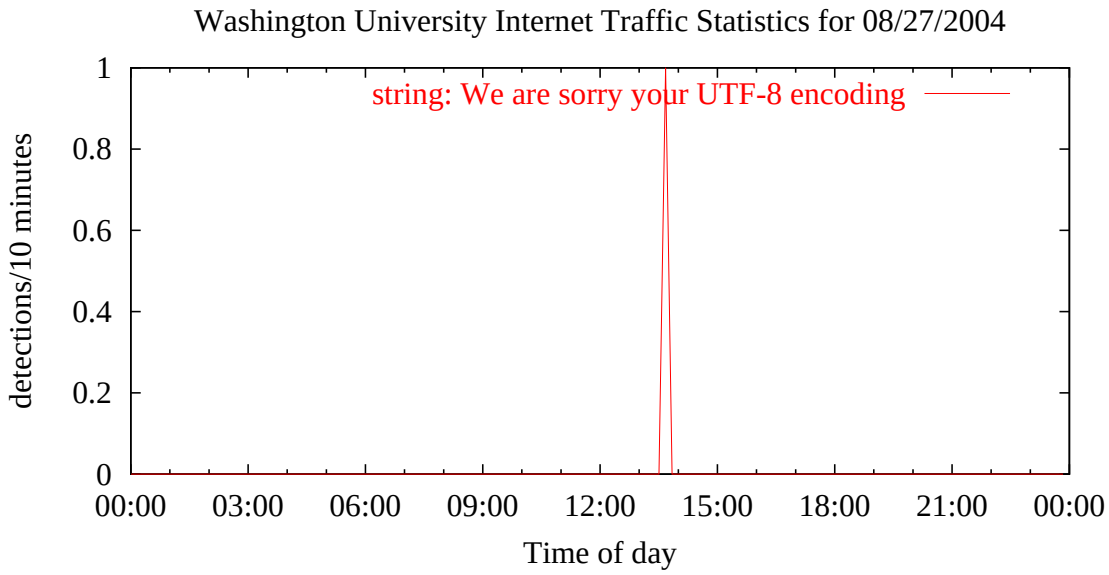


Figure D.25: MyDoom Virus 2

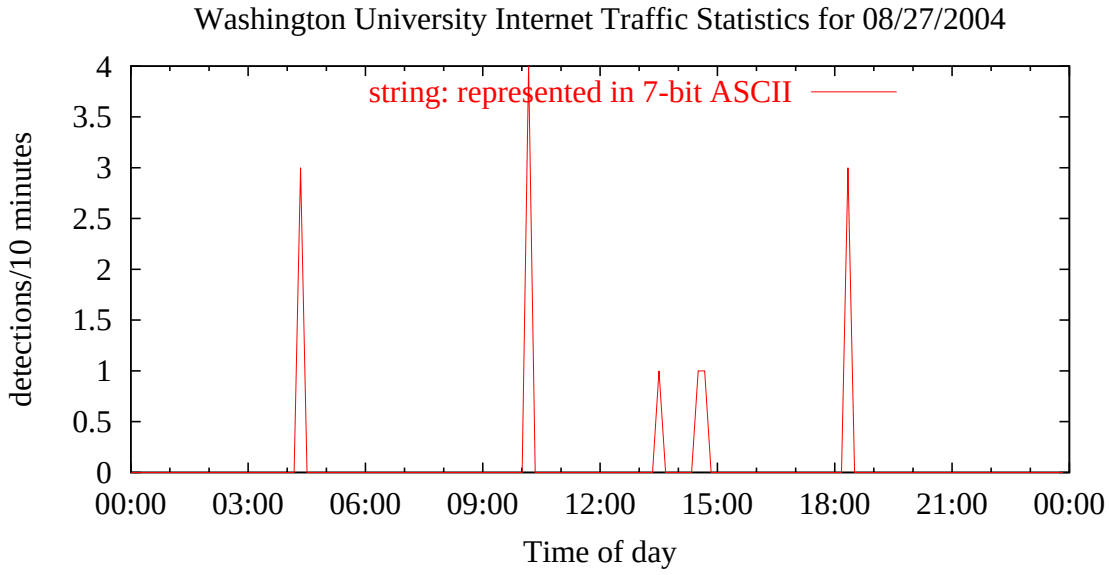


Figure D.26: MyDoom Virus 3

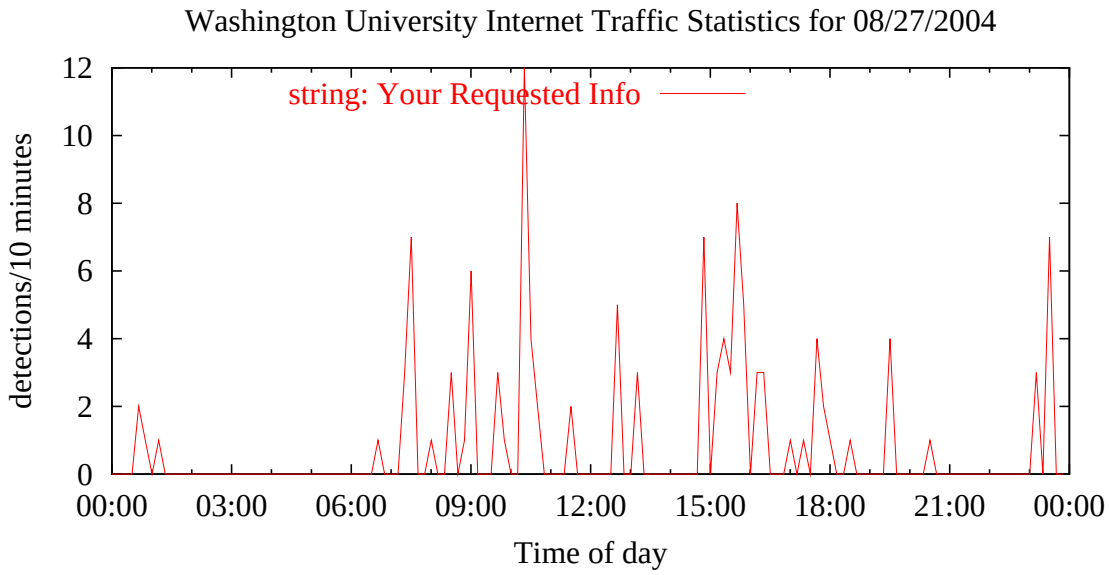


Figure D.27: Spam

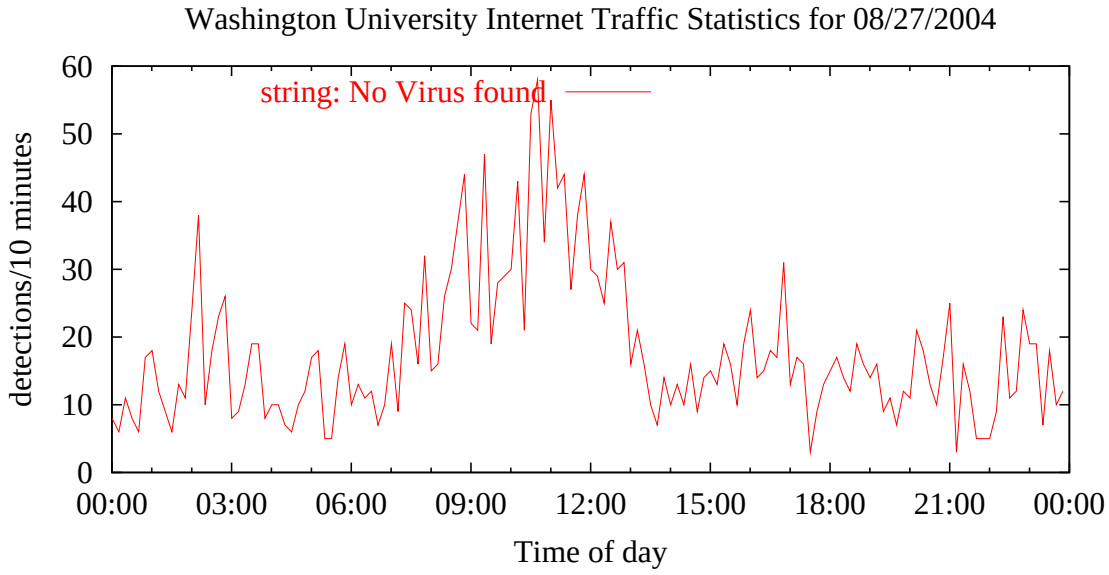


Figure D.28: Netsky Virus

D.2 Statistics for Sep 16, 2004

D.2.1 Flow Statistics

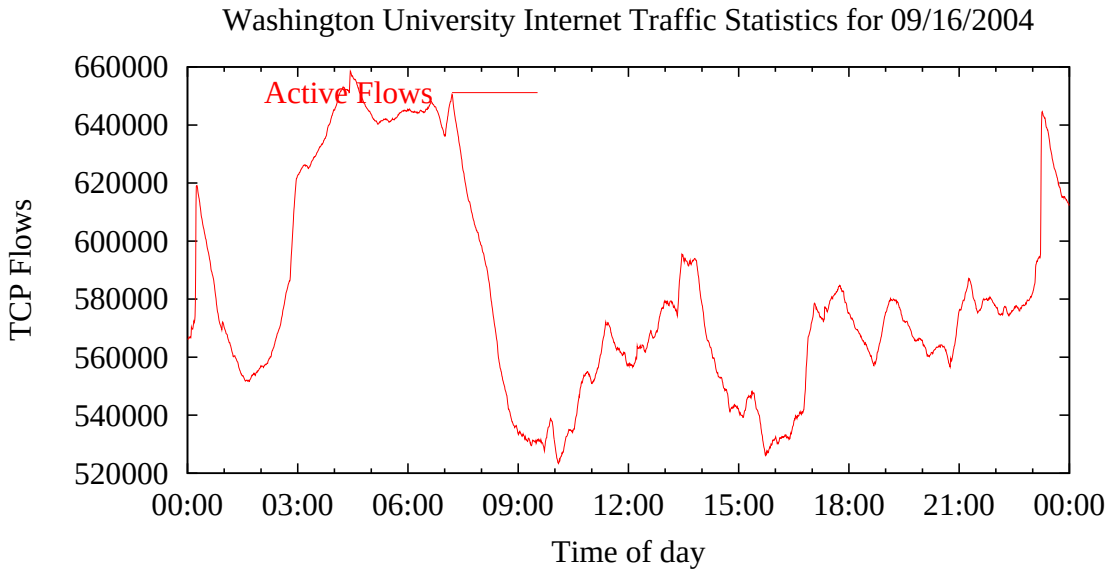


Figure D.29: Active Flows

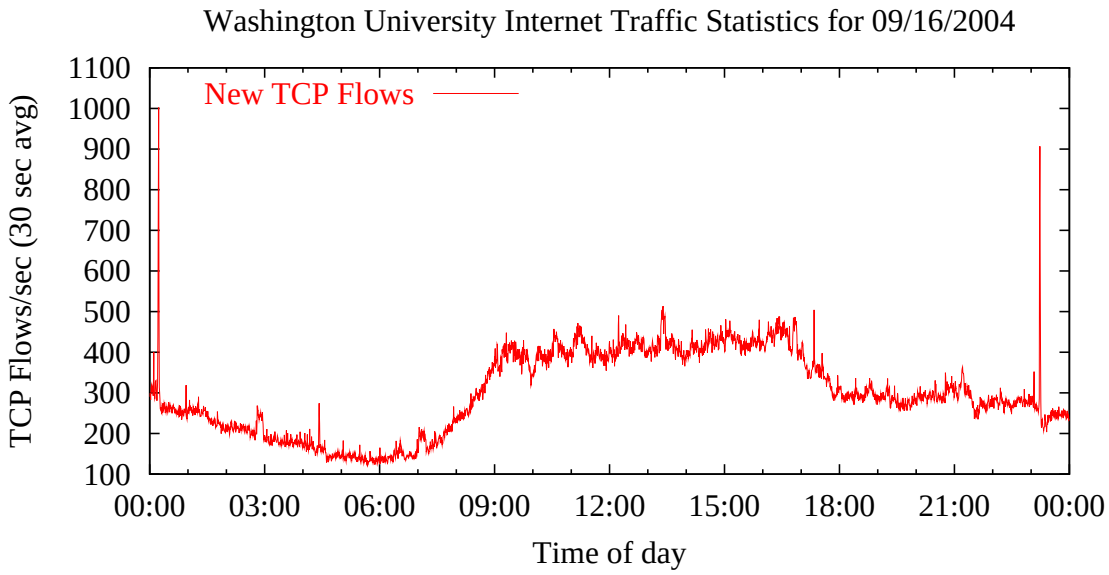


Figure D.30: New Flows

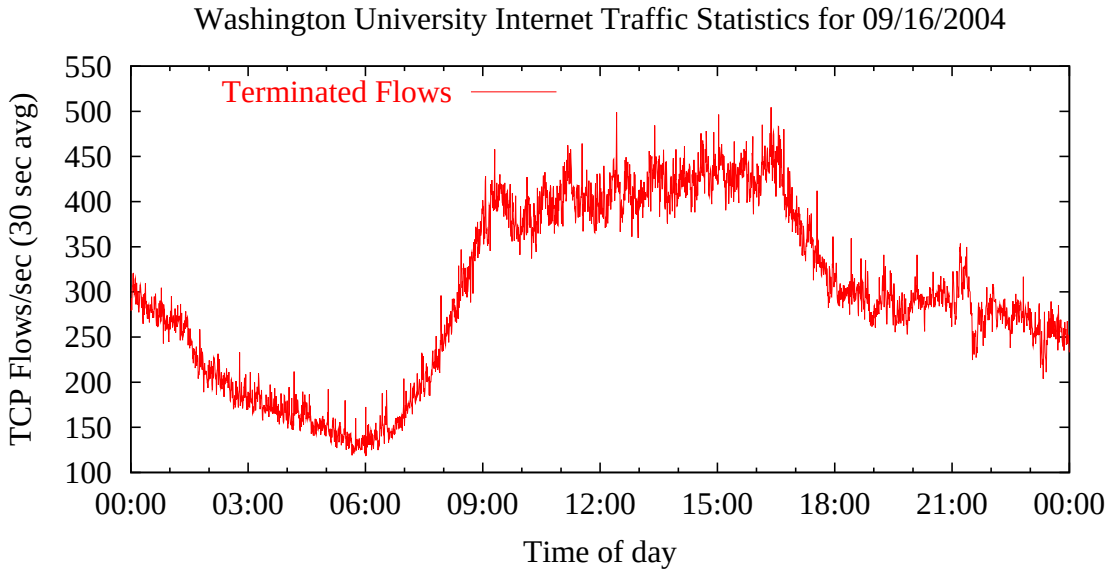


Figure D.31: Terminated Flows

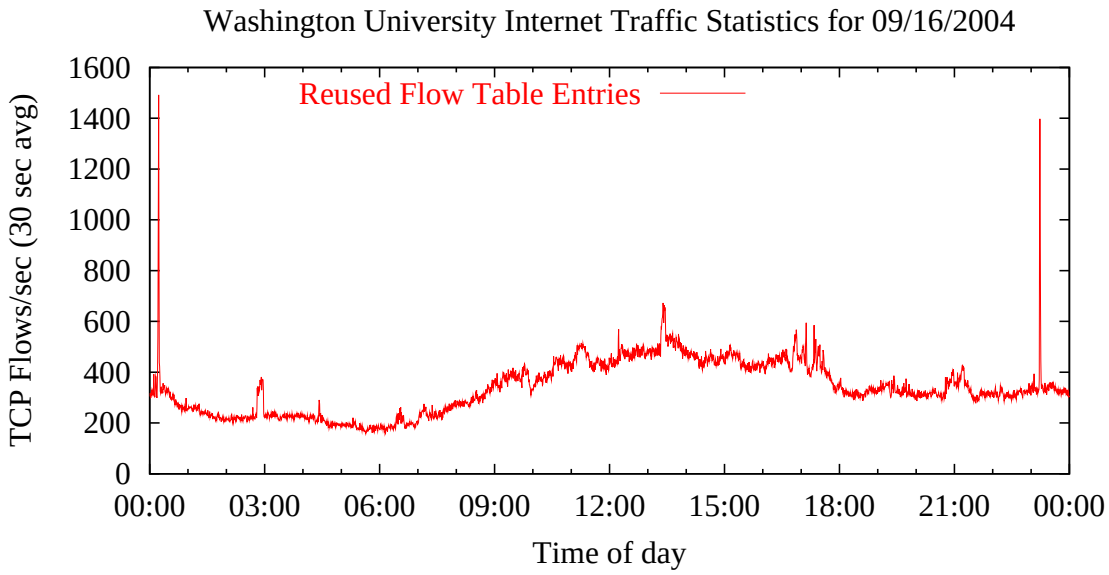


Figure D.32: Reused Flows

D.2.2 Traffic Statistics

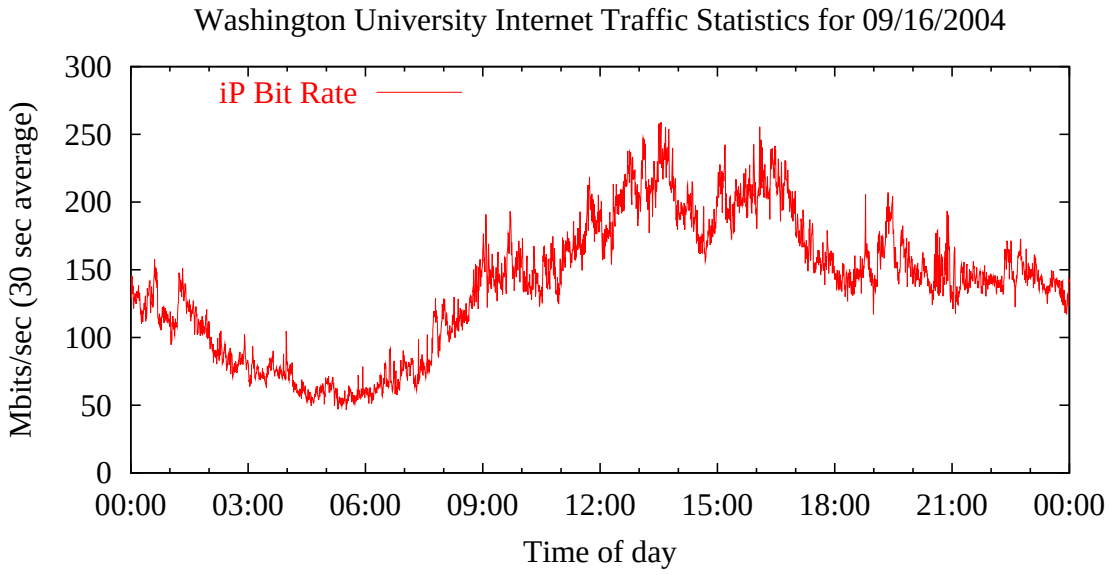


Figure D.33: IP bit rate

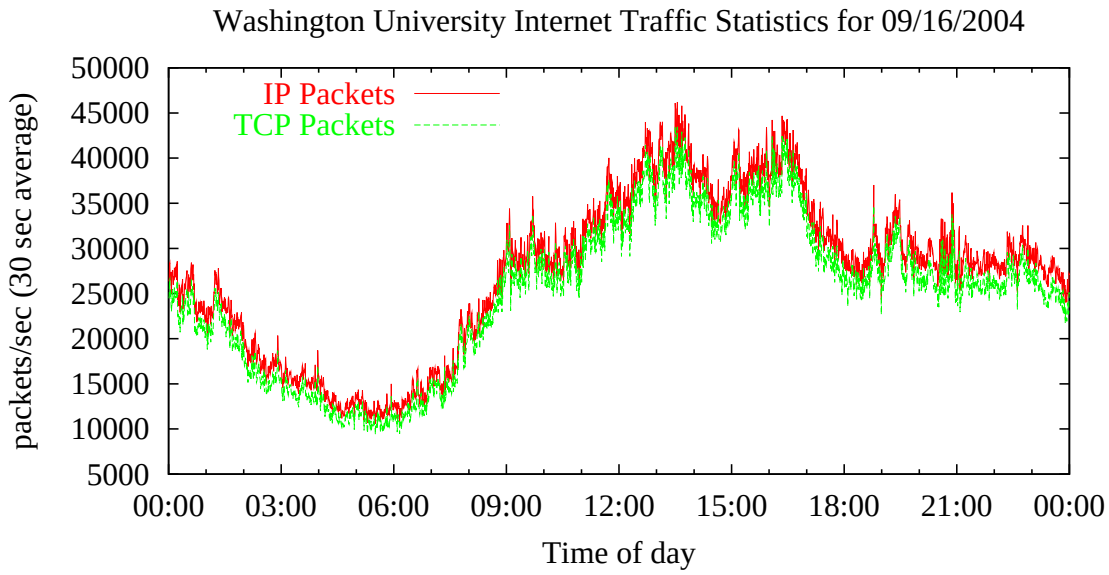


Figure D.34: IP Packets

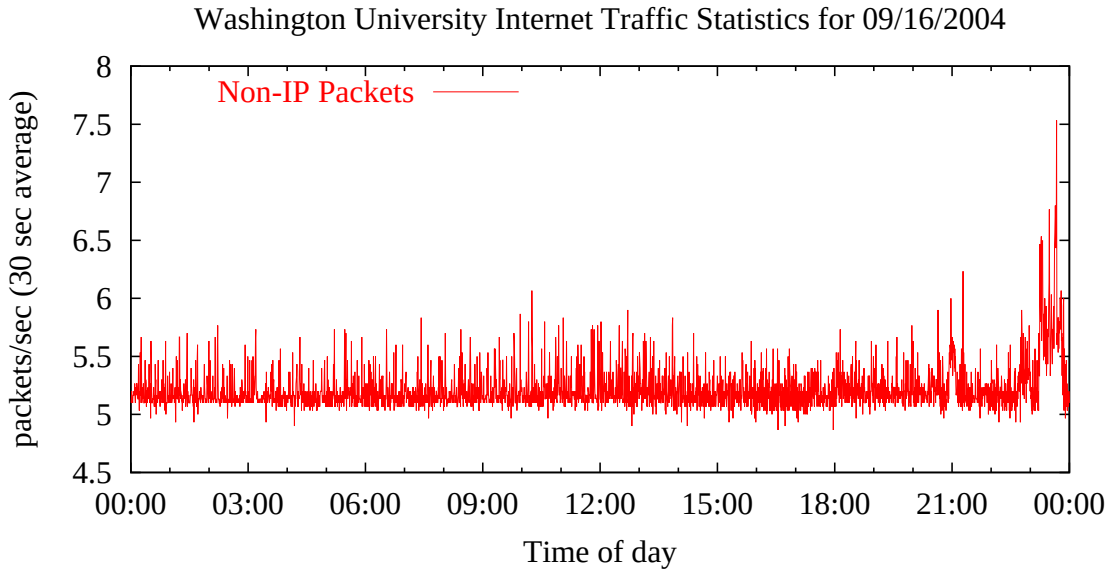


Figure D.35: Non-IP Packets

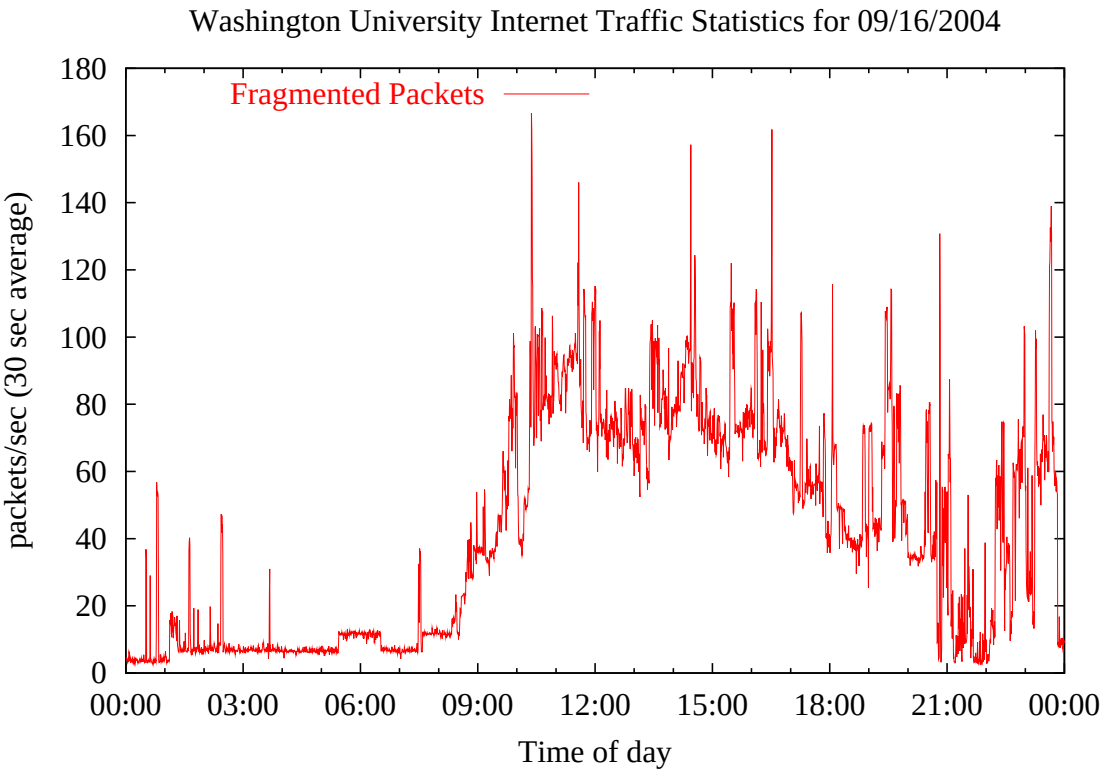


Figure D.36: Fragmented IP Packets

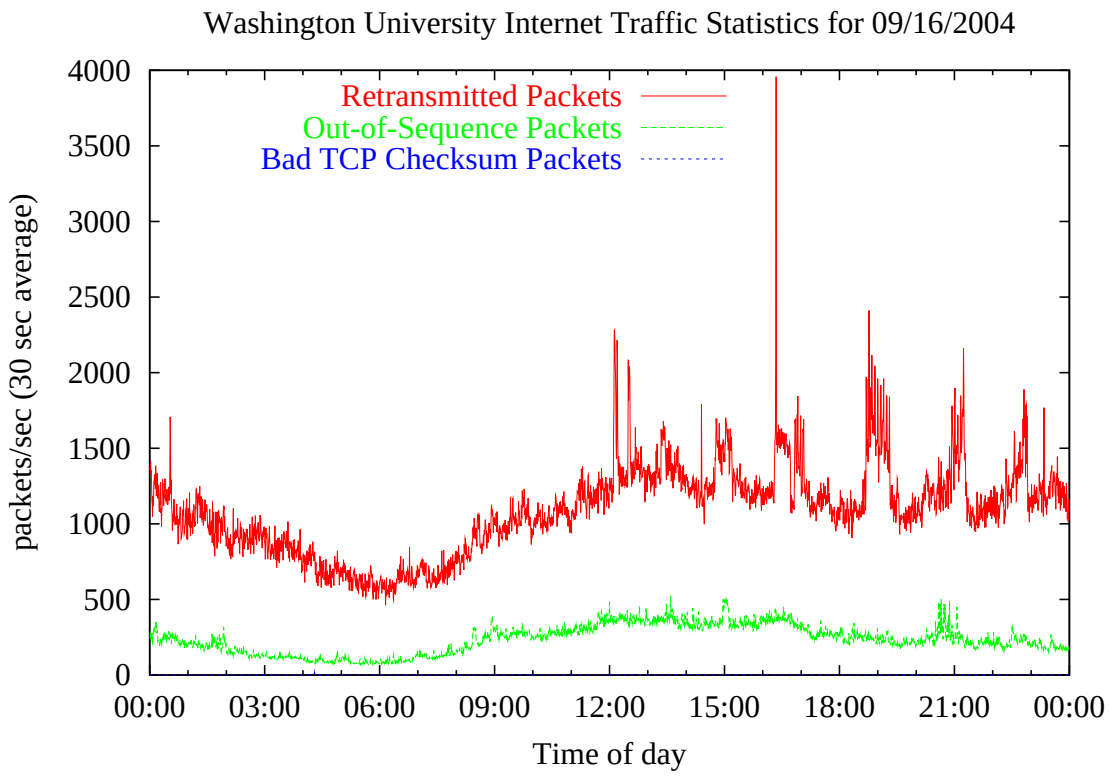


Figure D.37: Bad TCP Packets

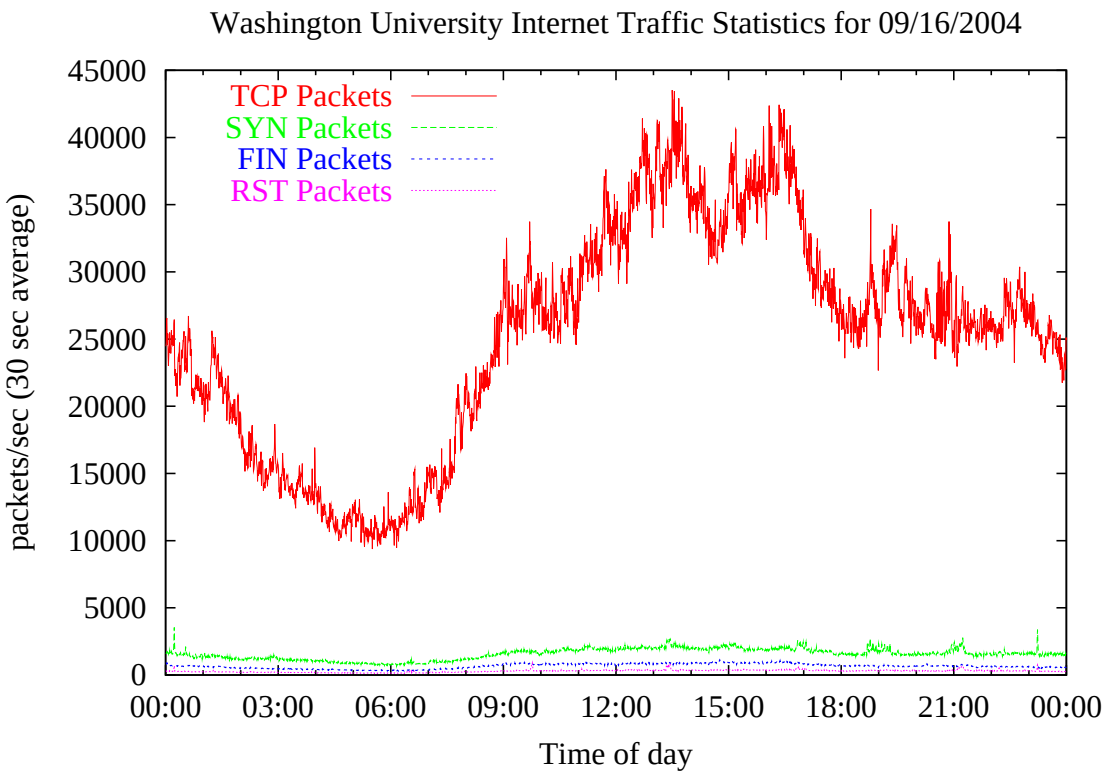


Figure D.38: TCP Packets

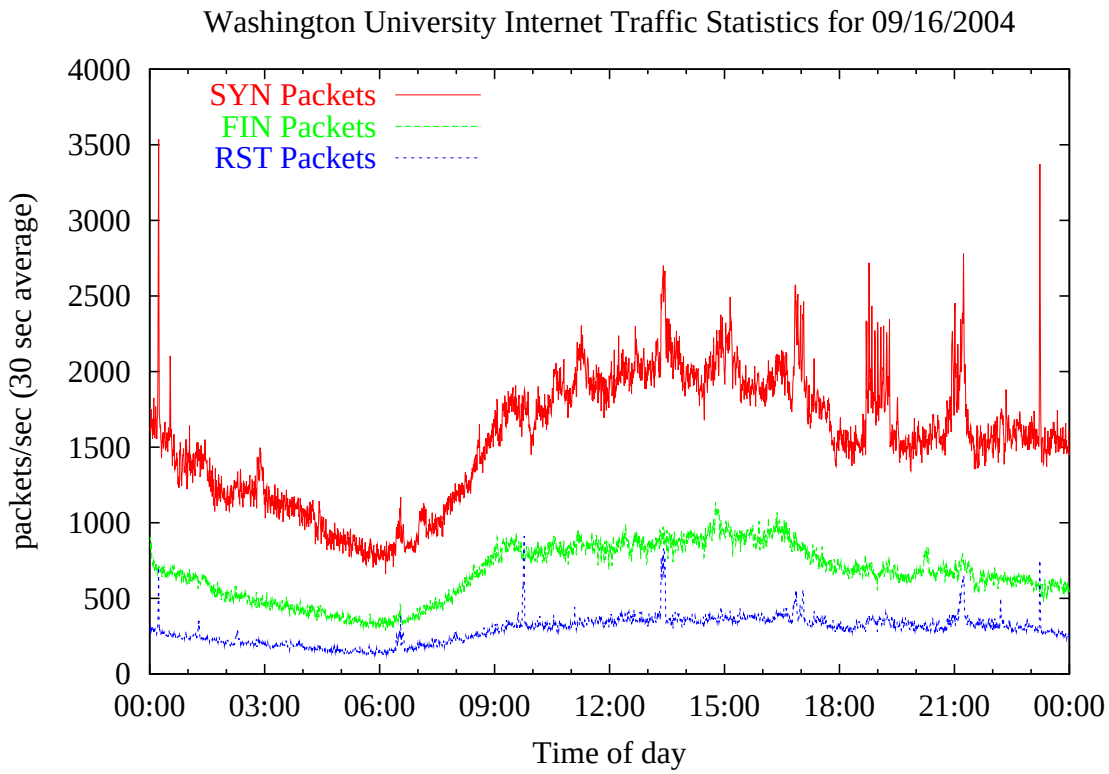


Figure D.39: TCP Packet Flags

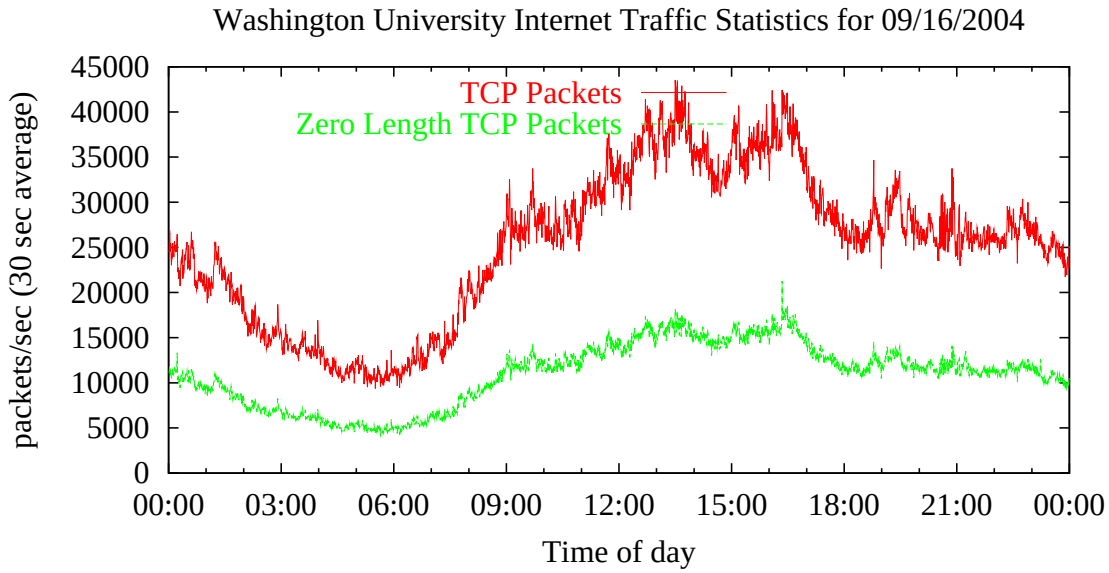


Figure D.40: Zero Length Packets

D.2.3 Port Statistics

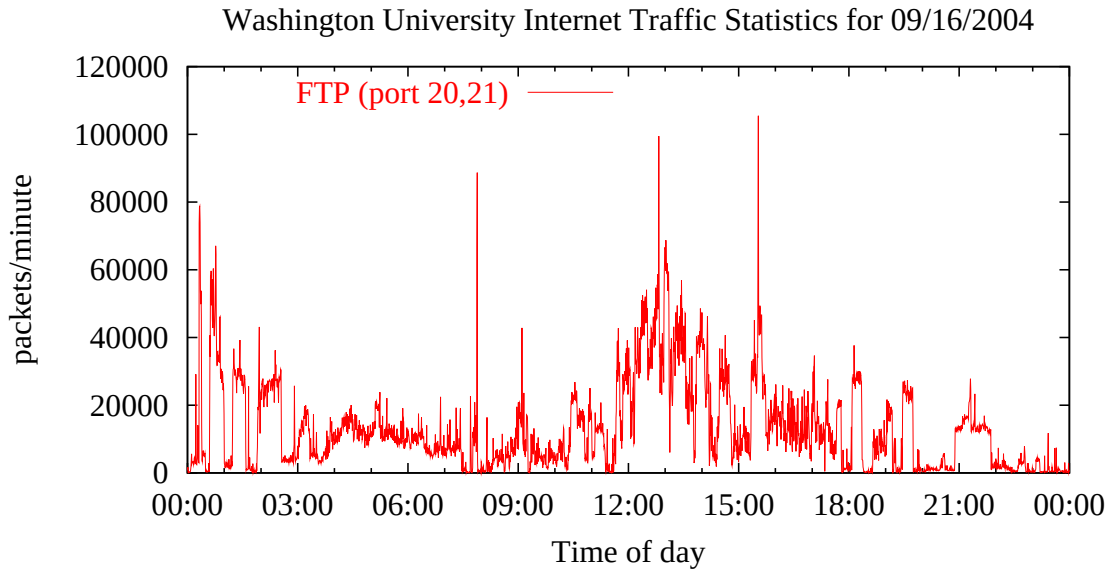


Figure D.41: FTP traffic

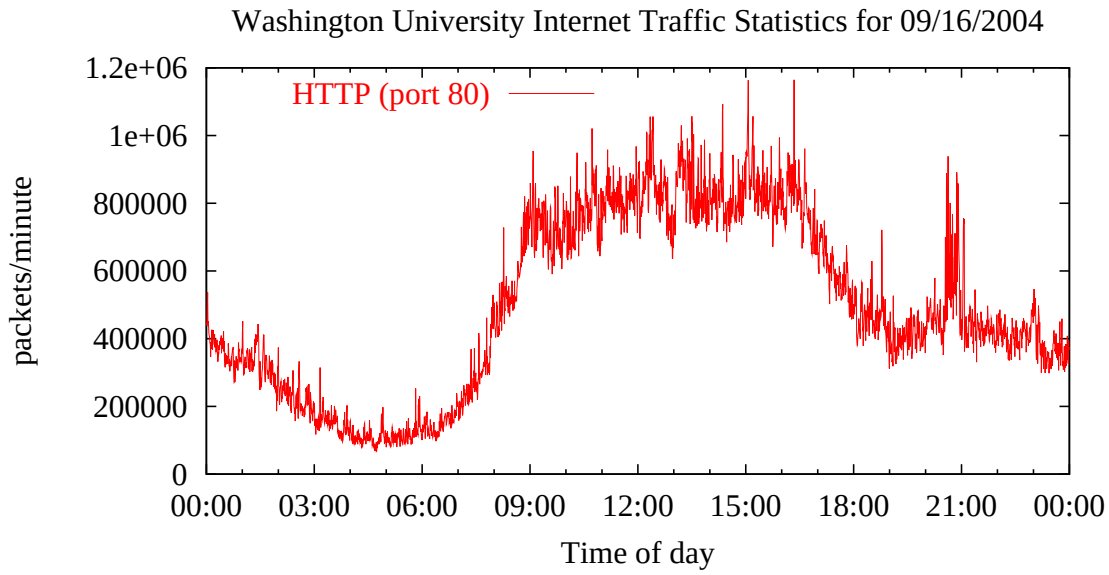


Figure D.42: HTTP traffic

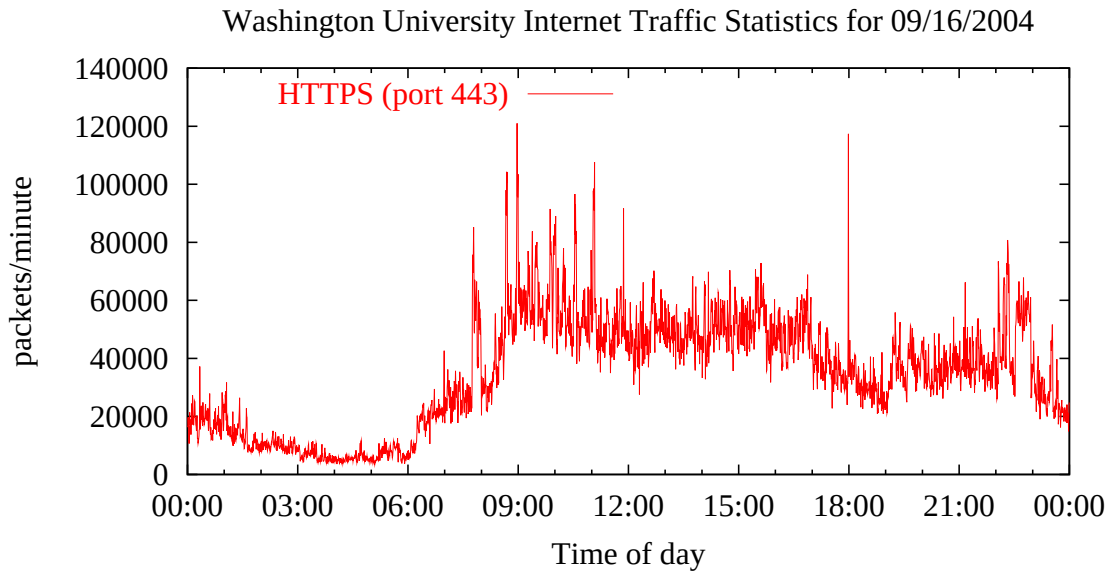


Figure D.43: HTTP traffic

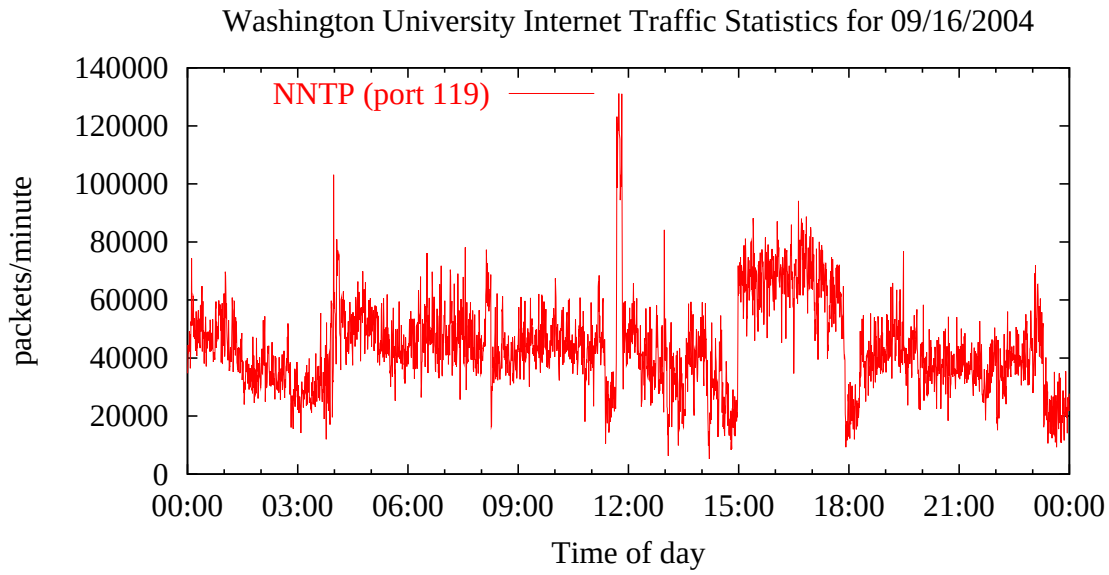


Figure D.44: NNTP traffic

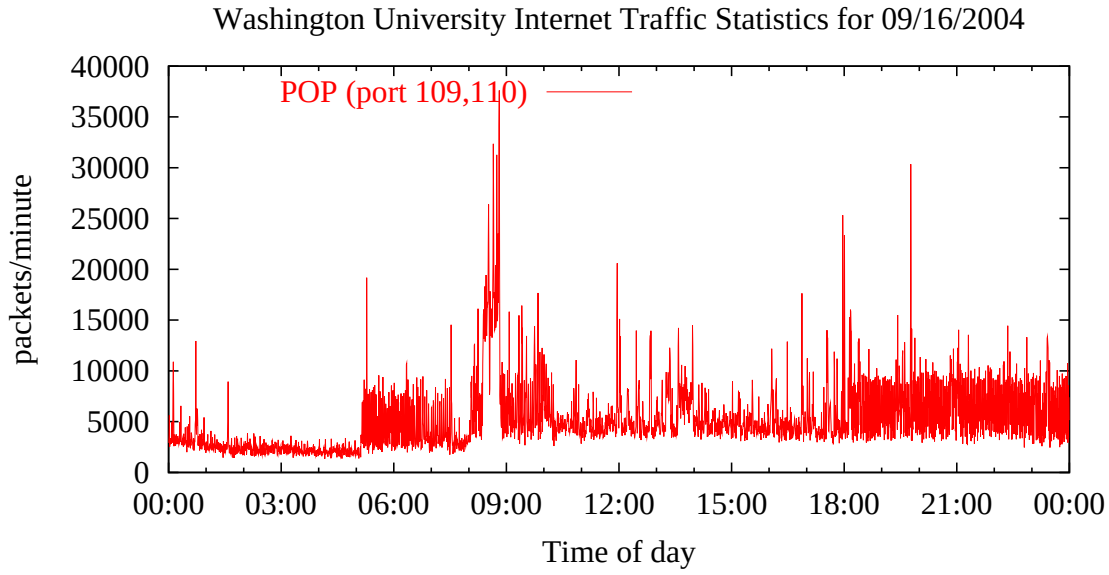


Figure D.45: POP traffic

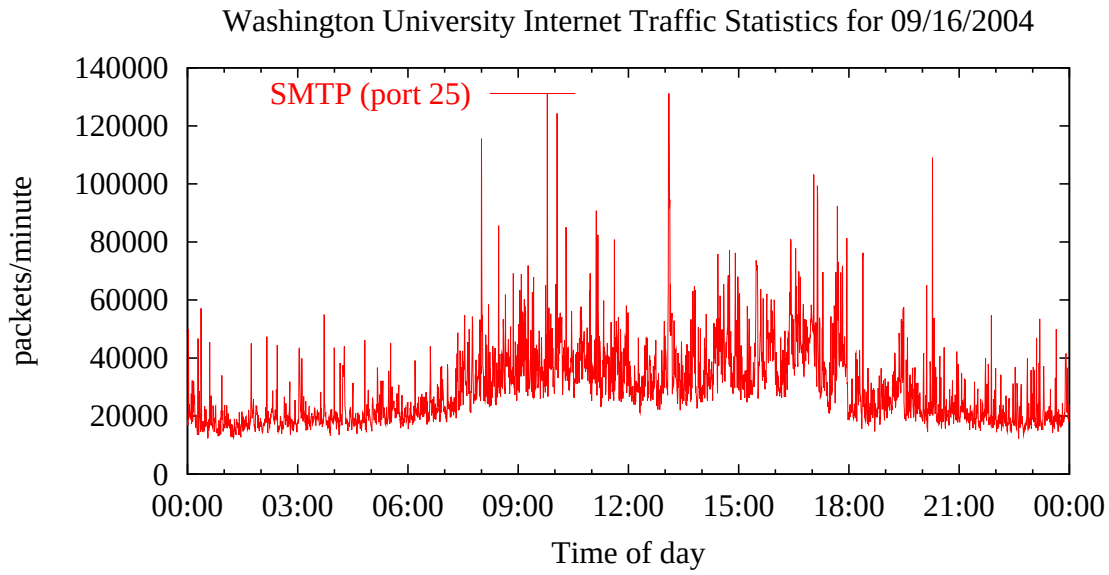


Figure D.46: SMTP traffic

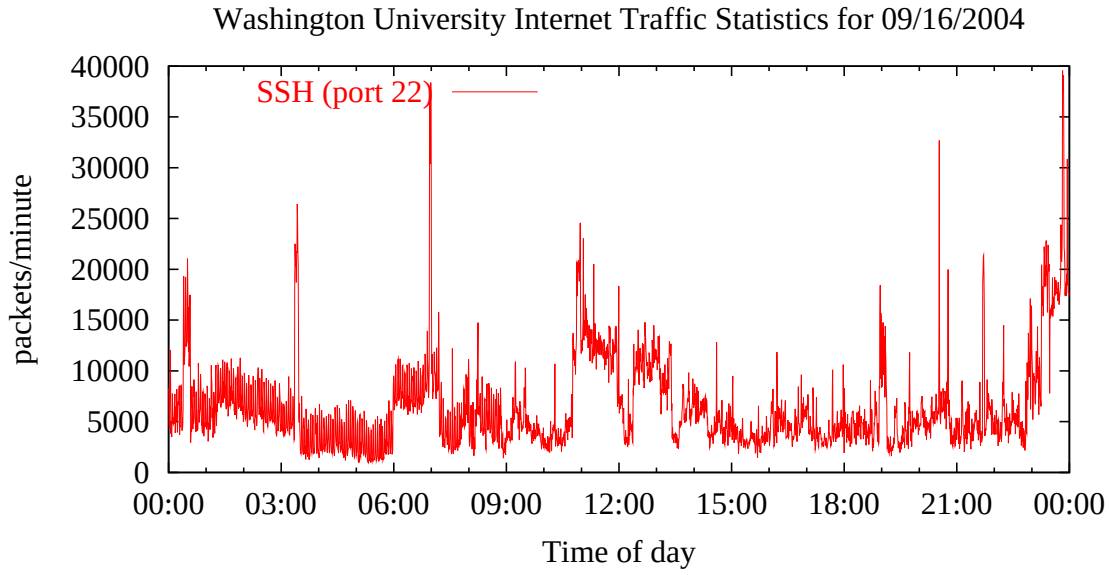


Figure D.47: SSH traffic

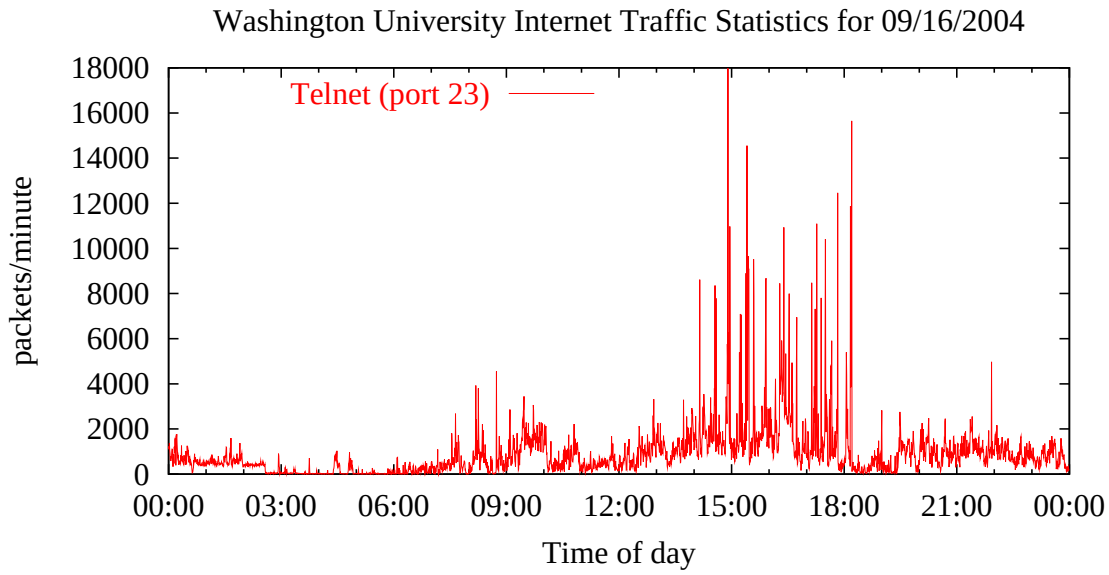


Figure D.48: Telnet traffic

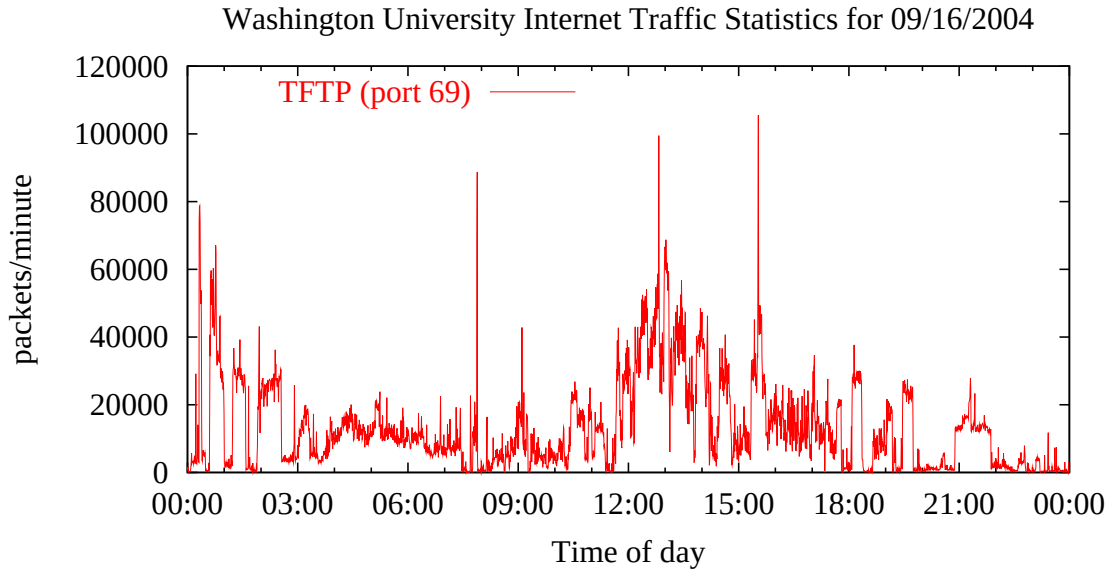


Figure D.49: TFTP traffic

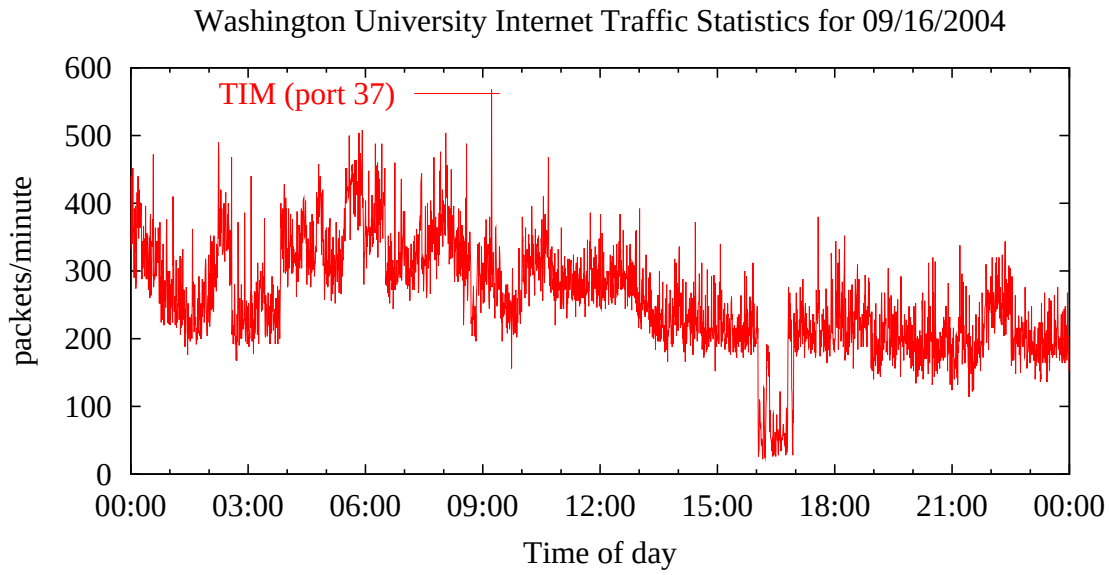


Figure D.50: TIM traffic

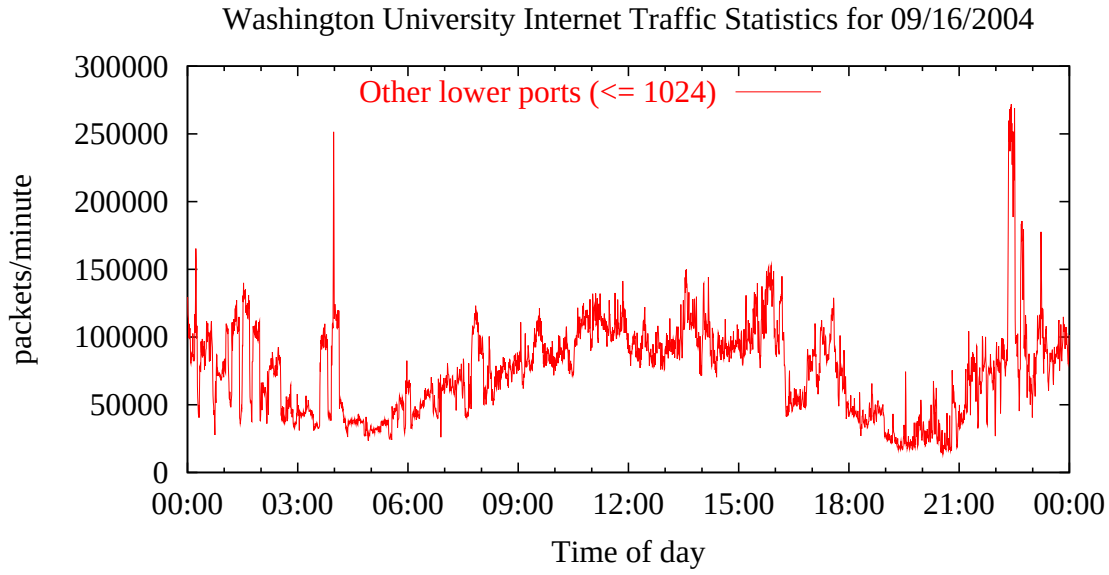


Figure D.51: Lower port traffic

D.2.4 Scan Statistics

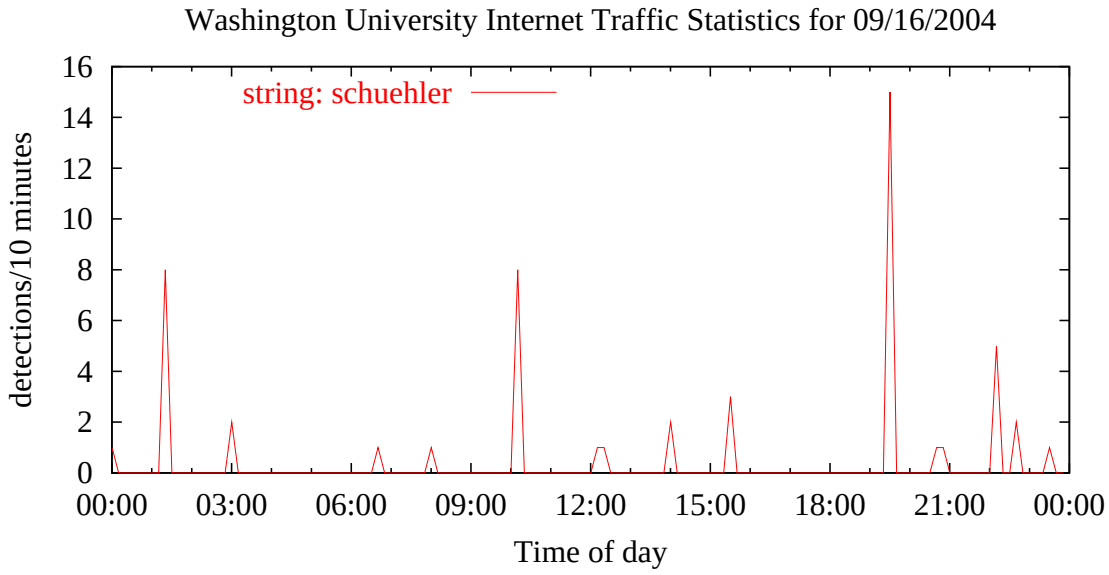


Figure D.52: Scan for HTTP

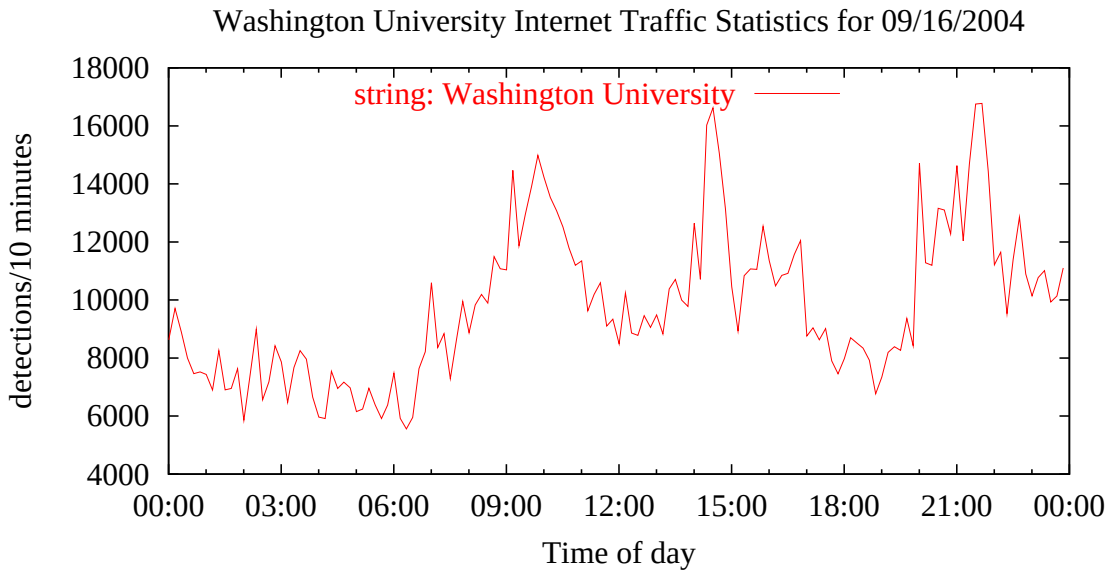


Figure D.53: Scan for Washington University

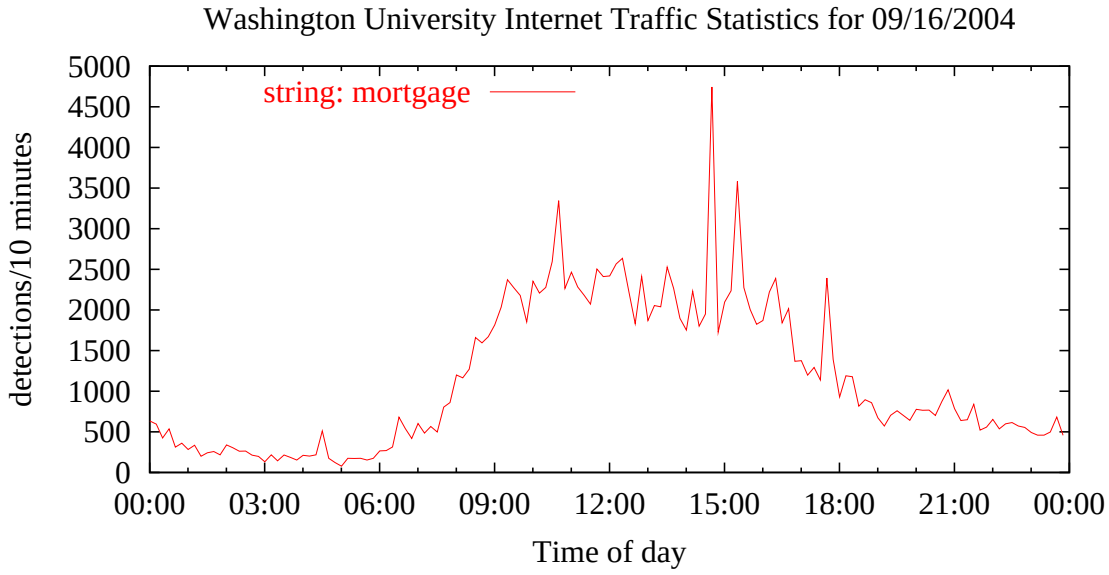


Figure D.54: Scan for mortgage

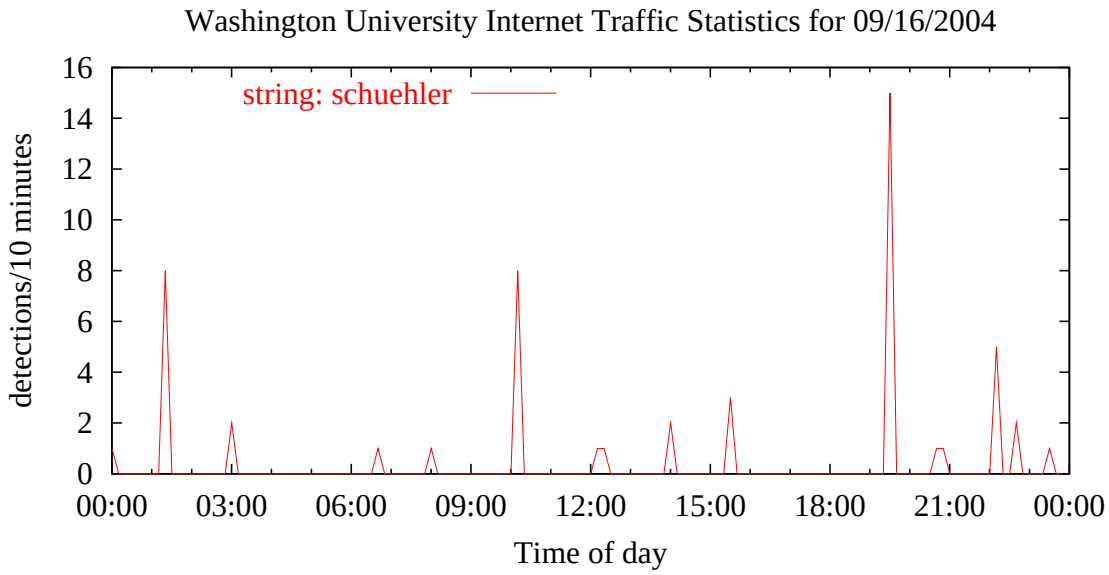


Figure D.55: Scan for schuehler

References

[1] Adam Piore. Hacking for Dollars. http://www.msnbc.msn.com/id/3706599, 2003.

[2] Nikos Anerousis, Ramon Caceres, Nick Duffield, Anja Feldmann, Albert Greenberg, Chuck Kalmanek, Partho Mishra, K.K. Ramakrishnan, and Jennifer Rexford. Using the AT&T Labs PacketScope for Internet Measurement, Design, and Performance Analysis. http://citeseer.nj.nec.com/477885.html, 1997.

[3] Florin Baboescu and George Varghese. Scalable packet classification. In *ACM SIGCOMM*, August 2001.

[4] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on fpgas. In *Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 223–232. ACM Press, 2004.

[5] BBC News. Modest cost of Sobig virus. http://news.bbc.co.uk/1/hi/business/3173243.stm, August 2003.

[6] Jon C. R. Bennett, Craig Partridge, and Nicholas Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking (TON)*, 7(6):789–798, 1999.

[7] F. Berman, G. Fox, and A. J. G. Hey. *Grid Computing - Making the Global Infrastructure a Reality*. John Wiley and Sons Ltd., 2003.

[8] Karthikeyan Bhargavan, Satish Chandra, Peter J. McCann, and Carl A. Gunter. What packets may come: automata for network monitoring. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 206–219. ACM Press, 2001.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[10] Jag Bolaria. *A Guide to Storage and TCP Processors*. The Linley Group, October 2002.

[11] James O. Bondi, Ashwini K. Nanda, and Simonjit Dutta. Integrating a misprediction recovery cache (mrc) into a superscalar pipeline. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 14–23. IEEE Computer Society, 1996.

[12] Florian Braun, John Lockwood, and Marcel Waldvogel. Reconfigurable router modules using network protocol wrappers. In *Proceedings of Field-Programmable Logic and Applications*, pages 254–263, Belfast, Northern Ireland, August 2001.

[13] Florian Braun, John W. Lockwood, and Marcel Waldvogel. Layered protocol wrappers for Internet packet processing in reconfigurable hardware. In *Proceedings of Symposium on High Performance Interconnects (HotI'01)*, pages 93–98, Stanford, CA, USA, August 2001.

[14] Florian Braun, John W. Lockwood, and Marcel Waldvogel. Layered protocol wrappers for Internet packet processing in reconfigurable hardware. Technical Report WU-CS-01-10, Washington University in Saint Louis, Department of Computer Science, June 2001.

[15] Bertrand R. Brinley. *Mad Scientists' Club*. Scholastic Book Services (republished by Purple House Press in 2004), 1965.

[16] Bertrand R. Brinley. *The New Adventures of the Mad Scientists' Club*. Scholastic Book Services (republished by Purple House Press in 2004), 1968.

[17] Young H. Cho, Shiva Navab, and William H. Mangione-Smith. Specialized hardware for deep network packet filtering. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, pages 452–461. Springer-Verlag, 2002.

[18] Cisco. CSS 11500 Series Content Services Switch. http://cisco.com/en/US/products/hw/contnetw/ps792/, 2004.

[19] Cisco. Intrusion Protection Data Sheet. http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/prodlit/netra_ds.pdf, 2004.

[20] Cisco. SCA 11000 Series Secure Content Accelerators. http://www.cisco.com/en/US/products/hw/contnetw/ps2083/, 2004.

[21] Christopher R. Clark and David E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications*, pages 956–959. Springer-Verlag, 2003.

[22] ComputerEconomics. Malicious Code Attacks Had $13.2 Billion Economic Impact in 2001. http://www.computereconomics.com/article.cfm?id=133, January 2002.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, pages 245–249. New Jersey: Prentice-Hall, 2001.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, page 252. New Jersey: Prentice-Hall, 2001.

[25] Andy Currid. Tcp offload to the rescue. *Queue*, 2(3):58–65, 2004.

[26] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM*, pages 3–14, 1997.

[27] John D. DeHart, William D. Richard, Edward W. Spitznagel, and David E. Taylor. The smart port card: An embedded Unix processor architecture for network management and active networking. Technical Report WUCS-01-18, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, August 2001.

[28] Robert Dewar and Matthew Smosna. *Microprocessors: A Programmer's View*. New York: McGraw-Hill, 1990.

[29] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep packet inspection using parallel bloom filters. In *Hot Interconnects*, pages 44–51, Stanford, CA, August 2003.

[30] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *SIAM Journal of Computing 23*, pages 738–761, 1994.

[31] William N. Eatherton. Hardware-based Internet protocol prefix lookups. Master's thesis, Washington University in Saint Louis, May 1999.

[32] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[33] Edward Hurley. Admins doubt arrests deter future worm writers. http://www.searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci922447,00.html, 2003.

[34] ExcelMicro. Email Virus Protection. http://www.excelmicro.com, 2004.

[35] F5. BIG-IP. http://f5.com/f5products/bigip/, 2004.

[36] F5. BIG-IP SSL Acceleration. http://www.f5.com/solutions/tech/security/ssl45.html, 2004.

[37] Anja Feldmann. BLT: Bi-Layer Tracing of HTTP and TCP/IP. *WWW9 / Computer Networks*, 33(1-6):321–335, 2000.

[38] Fortinet. FortiGate. http://fortinet.com/doc/FortinetBroch.pdf, 2004.

[39] Foundry. ServerIron. http://foundrynet.com/products/webswitches/serveriron/, 2004.

[40] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. *Lecture Notes in Computer Science*, 2170:556+, 2001.

[41] R. Franklin, D. Carver, and B. L. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, April 2002.

[42] Michael L. Fredman, Jnos Komls, and Endre Szemerdi. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[43] Alan Freier, Philip Karlton, and Paul Kocher. The SSL protocol, version 3. http://wp.netscape.com/eng/ssl3/draft302.txt, Nov 1996.

[44] George Gilder. *Telecosm*. Free Press, September 2000.

[45] Maya Gokhale, Dave Dubois, Andy Dubois, Mike Boorman, Steve Poole, and Vic Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In

Field Programmable Logic and Applications (FPL), pages 404–413, Montpellier, France, September 2002. Springer-Verlag.

[46] Ian Goldberg. Internet Protocol Scanning Engine. http://www.cs.berkeley.edu/ iang/isaac/ipse.html, 1996.

[47] Pankaj Gupta and Nick McKeown. Packet Classification on Multiple Fields. In *ACM SIGCOMM*, 1999.

[48] Pankaj Gupta and Nick McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects VII*, August 1999.

[49] Andrei Gurtov. Effect of delays on TCP performance. In *Proceedings of IFIP Personal Wireless Communications '2001*, Aug 2001.

[50] Y. Hoskote, V. Erraguntla, D. Finan, J. Howard, D. Vangal, V. Veeramachaneni, H. Wilson, J. Xu, and N. Borkar. A 10GHz TCP offload accelerator for 10Gbps Etherent in 90nm dual-VT CMOS. In *ISSCC*, February 2003.

[51] Ido Dubrawsky. Firewall Evolution - Deep Packet Inspection. www.securityfocus.com/printable/infocus/1716, 2003.

[52] IETF. RFC791: Internet Protocol. http://www.faqs.org/rfcs/rfc791.html, Sep 1981.

[53] IETF. RFC793: Transmission Control Protocol. http://www.faqs.org/rfcs/rfc793.html, Sep 1981.

[54] IETF. RFC7771: A border gateway protocol 4 (BGP-4). http://www.faqs.org/rfcs/rfc1771.html, Mar 1995.

[55] Intrusion. SecureNet Sensors. http://intrusion.com/products/snsensors.asp, 2004.

[56] J. Ward. Threats and Vulnerabilities. *JANET-CERT Security Conference*, 2002.

[57] V Jacobson and R Braden. RFC1072: TCP Extensions for Long-Delay Paths. http://www.faqs.org/rfcs/rfc1072.html, Oct 1988.

[58] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Measurement and classification of out-of-sequence packets in a tier-1 IP backbone. Technical Report CS Dept. Tech. Report 02-17, UMass, May 2002.

[59] Adam Johnson and Kenneth Mackenzie. Pattern Matching in Reconfigurable Logic for Packet Classification. In *ACM CASES*, 2001.

[60] Kieth Sklower. A Tree-Based Packet Routing Table for Berkely Unix. Technical report, University of California, Berkeley, 1993.

[61] D. E. Knuth, J. H. Morris, and V. B. Patt. Efficient string matching: An aid to bibliographic search. In *Communications of the ACM*, volume 18, pages 333–340, 1975.

[62] D. E. Knuth, J. H. Morris, and V. B. Patt. Fast pattern matching in strings. In *SIAM Journal of Computing*, volume 6, pages 323–350, 1977.

[63] Donald E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching*, pages 518–526. New York: Addison-Wesley Publishing Company, 1973.

[64] C. Kruegel, F. Valeur, G. Vigna, and R.A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–293, Oakland, CA, May 2002. IEEE Press.

[65] Harvey Ku, John W. Lockwood, and David V. Schuehler. TCP programmer for FPXs. Technical Report WUCS-2002-29, Washington University in Saint Louis, August 2002.

[66] Fred Kuhns, John DeHart, Ralph Keller, John Lockwood, Prashanth Pappu, Jyoti Parwatikar, Ed Spitznagel, William Richard, David Taylor, Jonathan Turner, and Ken Wong. Implementation of an open multi-service router. Technical Report WUCS-01-20, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, August 2001.

[67] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, Purdue, IN, 1995.

[68] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Internet routing instability. *IEEE/ACM Transactions on Networking*, 6(5):515–528, 1998.

[69] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *ACM SIGCOMM*, September 1998.

[70] Wenke Lee, Joao Cabrera, Ashley Thomas, Niranjan Balwalli, Sunmeet Saluja, and Yi Zhang. Performance adaptation in real-time intrusion detection systems. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, Lecture Notes in Computer Science, Berlin–Heidelberg–New York, October 2002. Springer-Verlag.

[71] Shaomeng Li, Jim Toressen, and Oddvar Soraasen. Exploiting reconfigurable hardware for network security. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, April 2003.

[72] Shaomeng Li, Jim Toressen, and Oddvar Soraasen. Exploiting stateful inspection of network security in reconfigurable hardware. In *Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, September 2003.

[73] John W Lockwood. Evolvable Internet hardware platforms. In *The Third NASA/DoD Workshop on Evolvable Hardware (EH'2001)*, pages 271–279, July 2001.

[74] John W Lockwood. An open platform for development of network processing modules in reprogrammable hardware. In *IEC DesignCon'01*, pages WB–19, Santa Clara, CA, January 2001.

[75] John W. Lockwood, James Moscola, Matthew Kulig, David Reddick, and Tim Brooks. Internet worm and virus protection in dynamically reconfigurable hardware. In *Military and Aerospace Programmable Logic Device (MAPLD)*, page E10, Washington DC, September 2003.

[76] John W. Lockwood, James Moscola, David Reddick, Matthew Kulig, and Tim Brooks. Application of hardware accelerated extensible network nodes for Internet worm and virus protection. In *International Working Conference on Active Networks (IWAN)*, Kyoto, Japan, December 2003.

[77] John W. Lockwood, Christopher Neely, Christopher Zuver, James Moscola, Sarang Dharmapurikar, and David Lim. An extensible, system-on-programmable-chip, content-aware Internet firewall. In *Field Programmable Logic and Applications (FPL)*, page 14B, Lisbon, Portugal, September 2003.

[78] John W. Lockwood, Jon S. Turner, and David E. Taylor. Field programmable port extender (FPX) for distributed routing and queuing. In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, pages 137–144, Monterey, CA, USA, February 2000.

[79] Udi Manber. *Introduction to Algorithms, A Creative Approach*, pages 78–80. New York: Addison-Wesley Publishing Company, 1989.

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

[81] McAfee. McAfee VirusScan. http://us.mcafee.com/root/package.asp?pkgid=100, 2004.

[82] Anthony J. McAuley and Paul Francis. Fast routing table lookup using CAMs. In *INFOCOM (3)*, pages 1382–1391, 1993.

[83] Steve McCanne, Craig Leres, and Van Jacobson. tcpdump. ftp://ftp.ee.lbl.gov, 1998.

[84] Jennifer Mears. Radware powers up SSL accelerator. *NetworkWorldFusion*, 2002.

[85] mi2g. Digital Attacks Report - SIPS Monthly Intelligence Description, Economic Damage - All Attacks - Yearly. http://www.mi2g.net/cgi/mi2g/sipsgraph.php, September 2004.

[86] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(04):33–39, 2003.

[87] David Moore and Colleen Shannon. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of ACM SIGCOMM '02*, pages 273–284, November 2002.

[88] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stephan Savage. Internet quarantine: Requirements for containing self-propagating code. In *IEEE Infocom 2003*, San Francisco, CA, March 2003.

[89] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[90] Donald R. Morrison. Patriciapractical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[91] Marc Necker, Didier Contis, and David Schimmel. TCP-Stream Reassembly and State Tracking in Hardware. FCCM 2002 Poster, Apr 2002.

[92] Nortel. Alteon Content Director. http://www.nortelnetworks.com/products/01/pcd/, 2004.

[93] Nortel. Alteon SSL Accelerator. http://www.nortelnetworks.com/products/01/alteon/isdssl/index.html, 2004.

[94] Gary J. Nutt. *Centralized and Distributed Operating Systems*, pages 13–15. New Jersey: Prentice-Hall, 1992.

[95] Angela Orebaugh, Greg Morris, Ed Warnicke, and Gilbert Ramirez. *Ethereal Packet Sniffing*. Syngress Publishing, 2004.

[96] Arne Oslebo. TCP Revisited. In *NORDUnet 2002*, April 2002.

[97] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer*, pages 247–256, 1990.

[98] Rina Panigrahy and Samar Sharma. Reducing TCAM Power Consumption and Increasing Throughput. In *Proceedings of Symposium on High Performance Interconnects (HotI'02)*, pages 107–111, Stanford, CA, USA, August 2002.

[99] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[100] Amit Prakash and Adnan Aziz. OC-3072 Packet Classification Using BDDs and Pipelined SRAMs. In *Proceedings of Symposium on High Performance Interconnects (HotI'01)*, pages 15–20, Stanford, CA, USA, August 2001.

[101] Radware. Web Server Director. http://www.radware.com/content/products/wsd/, 2004.

[102] Greg Regnier, Dave Minturn, Gary McAlpine, Vikram Saletore, and Annie Foong. ETA: Experience with and Intel Xeon Processor as a Packet Processing Engine. In *Hot Interconnects 11*, August 2003.

[103] H. Norton Riley. The von Neumann architecture of computer systems. http://www.csupomona.edu/ hnriley/www/VonN.html, 1987.

[104] Martin Roesch. SNORT - Lightweight Intrusion Detection for Networks. In *LISA '99: USENIX 13th Systems Administration Conference*, November 1999.

[105] Patrick Brooks Roland Wooster, Stephen Williams. HTTPDUMP Network HTTP Packet Snooper. http://citeseer.nj.nec.com/332269.html, Apr 1996.

[106] Ravi Sabhikhi. Network processor requirements for processing higher layer protocols such as TCP/IP. http://www.cs.washington.edu/NP2/ravi.s.invited.talk.pdf, Feb 2003.

[107] Lambert Schaelicke, Thomas Slabach, Branden Moore, and Curt Freeland. Characterizing the performance of network intrusion detection sensors. In *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, Lecture Notes in Computer Science, Berlin–Heidelberg–New York, September 2003. Springer-Verlag.

[108] David V. Schuehler, Harvey Ku, and John Lockwood. A TCP/IP based multi-device programming circuit. In *Field Programmable Logic and Applications (FPL)*, page P2.B, Lisbon, Portugal, September 2003.

[109] David V. Schuehler and John Lockwood. TCP-Splitter: Design, implementation, and operation. Technical Report WUCSE-2003-14, Washington University in Saint Louis, March 2003.

[110] David V. Schuehler and John W. Lockwood. Tcp-splitter: A TCP/IP flow monitor in reconfigurable hardware. In *Proceedings of Symposium on High Performance Interconnects (HotI'02)*, pages 127–131, Stanford, CA, USA, August 2002.

[111] Devavrat Shah and Pankaj Gupta. Fast incremental updates on Ternary-CAMs for routing lookups and packet classification. In *Proceedings of Symposium on High Performance Interconnects (HotI'00)*, Stanford, CA, USA, August 2000.

[112] Stanislav Shalunov and Benjamin Teitelbaum. Bulk TCP use and performance on Internet2. In *Proceedings of ACM SIGCOMM Measurement Workshop*, Aug 2001.

[113] Sajjan G Shiva. *Pipelined and Parallel Computer Architectures*, pages 44–45. New York: Harper Collins, 1996.

[114] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2001.

[115] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, pages 213–224, Aug 2003.

[116] J. E. Smith and G. S. Sohi. The microarchitecture of superscaler processors. In *Proceedings of the IEEE*, December 1995.

[117] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 262–271. ACM Press, 2003.

[118] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications*, pages 880–889. Springer-Verlag, 2003.

[119] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. Fast and scalable layer four switching. In *ACM SIGCOMM*, June 1998.

[120] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Trans. Comput. Syst.*, 17(1):1–40, 1999.

[121] Clifford Stoll. *Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Pocket (republished in 2000), 1990.

[122] Yutaka Sugawara, Mary Inaba, and Kei Hiraki. Over 10gbps string matching mechanism for multi-stream packet scanning systems. In *Field Programmable Logic and Applications (FPL)*, pages 484–493, Antwerp, Belbium, September 2004.

[123] Symantec. Norton AntiVirus. http://www.symantec.com/nav/nav_9xnt/, 2004.

[124] David E. Taylor, John W. Lockwood, and Sarang Dharmapurikar. Generalized RAD module interface specification of the field-programmable port extender (fpx). Technical Report WUCS-TM-01-15, Applied Research Laboratory, Department of Computer Science, Washington University in Saint Louis, July 2001. Available on-line as `http://www.arl.wustl.edu/arl/projects/fpx/wugs.ps`.

[125] David E. Taylor, John W. Lockwood, Todd S. Sproull, Jonathan S. Turner, and David B. Parlour. Scalable IP lookup for programmable routers. In *IEEE Infocom 2002*, New York NY, June 2002.

[126] David E. Taylor, John W. Lockwood, Todd S. Sproull, Jonathan S. Turner, and David B. Parlour. Scalable IP lookup for Internet routers. *IEEE Journal on Selected Areas in Communications*, 21(4), May 2003.

[127] K. Thompson, G. J. Miller, and R. Wilder. Wide-area Internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, (Nov/Dec 1997).

[128] David E. Taylor Todd Sproull, John W. Lockwood. Control and Configuration Software for a reconfigurable Networking Hardware Platform. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, April 2002.

[129] J. Turner, T. Chaney, A. Fingerhut, and M. Flucke. Design of a Gigabit ATM Switch. In *In Proceedings of Infocom 97*, March 1997.

[130] Jon Turner. Open network laboratory: A resource for networking researchers. http://www.arl.wustl.edu/arl/projects/onl/, 2004.

[131] Jon Turner. Technologies for dynamically extensible networks. http://arl.wustl.edu/projects/techX/, Jun 2004.

[132] United States. The national strategy to secure cyberspace. http://www.whitehouse.gov/pcipb/cyberspace_strategy.pdf, Feb 2003.

[133] G. Vigna, W. Robertson, V. Kher, and R.A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, pages 34–43, Las Vegas, NV, December 2003.

[134] Giovanni Vigna, Fredrik Valeur, and Richard A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–97. ACM Press, 2003.

[135] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high-speed prefix matching. *ACM Transactions on Computer Systems*, 19(4), November 2001.

[136] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[137] Xilinx. Virtex-E 1.8V FPGA Data Sheet. http://www.xilinx.com/bvdocs/publications/ds022-2.pdf, June 2004.

[138] Jianping Xu, Nitin Borkar, Vasantha Erraguntla, Yain Hoskote, Tanay Karnik, Sriram Vangal, and Justin Rattner. A 10Gbps Etherenet TCP/IP Processor. In *Hot Chips 15*, August 2003.

[139] Sudhakar Yalamanchili. *Introductory VHDL From simulation to syntehsis*, pages 36–43. New Jersey: Prentice-Hall, 2001.

[140] Cliff Changchun Zou, Lixin Gao, Weibo Gong, and Don Towsley. Monitoring and early warning for Internet worms. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 190–199. ACM Press, 2003.

Vita

David Vincent Schuehler

Date of Birth August 10, 1965

Place of Birth Cincinnati, Ohio

Degrees
D.Sc. Computer Engineering, December 2004
M.S. Computer Science, May 1993
B.S. Aeronautical and Astronautical Engineering, May 1988

Professional Societies
Association for Computing Machines
Institute of Electrical and Electronics Engineers

Publications

David Schuehler, Benjamin Brodie, Roger Chamberlain, Ron Cytron, Scott Friedman, Jason Fritts, Phillip Jones, Praveen Krishnamurthy, John Lockwood, Shobana Padmanabhan, Huakai Zhang. Microarchitecture Optimization for Embedded Systems, *High Performance Embedded Computing 8* (September 2004).

David V. Schuehler and John Lockwood. A Modular System for FPGA-based TCP Flow Processing in High-Speed Networks, *Field Programmable Logic and Applications 14* (August 2004).

Phillip Jones, Shobana Padmanabhan, Daniel Rymarz, John Maschmeyer, David V. Schuehler, John Lockwood, and Ron Cytron. Liquid Architecture, *International Parallel and Distributed Processing Symposium* (April 2004).

David V. Schuehler, Harvey Ku, and John Lockwood. A TCP/IP Based Multi-Device Programming Circuit, *Field Programmable Logic and Applications 13* (August 2003).

David V. Schuehler, James Moscola, and John Lockwood. Architecture for a Hardware Based, TCP/IP Content Scanning System, *Hot Interconnects 11* (August 2003) pgs: 98–94.

David V. Schuehler and John Lockwood. TCP-Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware, *Hot Interconnects 10* (August 2002) pgs: 127–131.

Journals

David V. Schuehler, James Moscola, and John Lockwood. Architecture for a Hardware Based, TCP/IP Content Scanning System, *IEEE Micro* (January/February 2004) **24**(1): 62–69.

David V. Schuehler and John Lockwood. TCP-Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware, *IEEE Micro* (January/February 2003) **23**(1): 54–59.

Awards

St. Louis Business Journal 2004 Technology Award, *St. Louis Business Journal* (April 2004).

3rd Place Award, *Washington University Graduate Student Research Symposium* (April 2004).

Technical Reports

David V. Schuehler TCP-Processor: Design, Implementation, Operation and Usage, *WUCSE-2004-53* (September 2004).

David V. Schuehler and John Lockwood. TCP-Splitter: Design, Implementation, and Operation, *WUCSE-2003-14* (March 2003).

Harvey Ku, David V. Schuehler, and John Lockwood. TCP Programmer for FPXs, *WUCS-2002-29* (August 2002).

Patents

David V. Schuehler and John W. Lockwood. Reliable packet monitoring methods and apparatus for high speed networks (Filed: August 2003).

David V. Schuehler and John W. Lockwood. TCP-Splitter: Reliable packet monitoring methods and apparatus for high-speed networks, (Filed: August 2002).

Industry Experience

Financial Industry

Technical Vice President - Research & Development
June 1991 to present
Perform various activities associated with delivering real-time financial data to the global community over a worldwide IP based network. Develop embedded software, device drivers, communications software, and application software in Assembly, Pascal, C, C++, Java and various scripting languages to process and deliver financial data worldwide. Design, architect, implement, support, and manage major components of the data delivery system.

Defense Industry

Senior Software Engineer
June 1989 to June 1991
Develop real-time software in FORTRAN, C and Assembly to perform hardware-in-the-loop missile simulations. The tasks ranged from developing software to generate, control and manipulate NTSC video that interfaced with flight hardware, firmware for interface electronics which sit between flight hardware and main simulation computers, telemetry processing software, and 3-D flight visualization software.

Healthcare Industry

System Programmer
October 1988 to June 1989
Maintain, support and manage computer equipment for 1,000 bed hospital. Develop client-server based applications, communications software, and various utilities for medical records and operations departments. Make recommendations for equipment purchases, perform software installations and upgrades, train users and troubleshoot problems.