Performance Analysis of TCAMs in Switches

by

Abdel Maguid Tawakol

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

Waterloo, Ontario, Canada, 2012

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

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

Abstract

The Catalyst 6500 is a modern commercial switch, capable of processing millions of packets per second through the utilization of specialized hardware. One of the main hardware components aiding the switch in performing its task is the Ternary Content Addressable Memory (TCAM). TCAMs update themselves with data relevant to routing and switching based on the traffic flowing through the switch. This enables the switch to forward future packets destined to a location that has already been previously discovered - at a very high speed.

The problem is TCAMs have a limited size, and once they reach their capacity, the switch has to rely on software to perform the switching and routing - a much slower process than performing Hardware Switching that utilizes the TCAM. A framework has been developed to analyze the switch's performance once the TCAM has reached its capacity, as well as measure the penalty associated with a cache miss. This thesis concludes with some recommendations and future work.

Acknowledgements

All praises are due to Allah, the most merciful, the most gracious. I thank Allah for blessing me, and giving me the ability to complete this work to the best of my ability.

I would like to thank my supervisor, Professor Gordon B. Agnew, for the support in selecting a topic for my Masters, and for sharing his deep insight on the topic. I would also like to thank Seyed Ali Ahmadzadeh for his support and guidance in navigating through this project.

I would like to thank the Cisco Engineers Ramesh Santhanakrishnan, Paolo Zarpellon, and Shayam Kapadia for their support and commitment to this project. Their help and wealthy knowledge on the design, implementation, and operation of the Catalyst 6500, has been essential for the completion of this project. I would like to thank my manager Suran De Silva for giving me the opportunity to work with some of the brightest minds in the industry, and gain extremely valuable work experience.

I will, forever, be indebted to my parents for their unparalleled support, encouragement, prayer, and love - I hope to always make you proud. Thank you for your help in getting to where I am today, and forging me into the person that I have become - I am truly blessed to have you in my life.

Dedication

To my beloved parents, Hashem and Suhair Tawakol.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet has become an integral part of modern society. Individuals from different backgrounds use the Internet to accomplish a large number of tasks which can vary from doing on-line banking, to accessing on-demand entertainment such as media streaming. While the activities can vary from user to user, the common demand for most activities is the need for fast and highly reliable access. In order to accomplish this demand, a strong infrastructure compromising of advanced hardware is required: high bandwidth cables and links, fast dedicated servers for communicating with end users, and advanced software techniques to deal with the process of routing and switching packets travelling across the Internet. One of the integral parts of the infrastructure are routers and switches. Routers and switches are hardware devices, that run specialized software, responsible for making sure that packets reach their destination.

1.1 Routers and Switches

Routers are devices responsible for connecting computer networks together for the purpose of data packet forwarding. Routers perform Layer 3 forwarding, also know as the network layer. This forwarding is made based on the IP address of the packet - a numerical label used to identify devices on the network. A router typically contains two or more ports, and contains a controller responsible for making packet routing decisions. The router is able to make a forwarding decision based on two pieces of information: the address information available in the packet's header, and the information available in the its routing table. The information in the routing table is either programmed or learned through some route discovery algorithm. When a packet arrives to the router, the header information

is extracted, and a lookup is performed in the routing table. Based on the information stored in the routing table, a decision is made on where this packet should go, and the packet is sent out on the appropriate port. In its simplest form, a router could be used to share an internet connection at home, where multiple computers are sharing the Internet service connected to an individual's home. Routers vary in bandwidth capabilities, security features, and implementable networking policies.

A network switch, also know as a switching hub, is a multi-port network bridge that processes and routes data at the data link layer (Layer 2). Some switches, know as multilayer switches, are capable of processing data at the network layer. Switches are able of performing Layer 2 (Data link layer) and Layer 3 (Network layer) switching, but often work at Layer 2. Another difference between routers and switches is that each type of device, uses a different algorithm for forwarding packets.

Switches contain a cache to store information helpful for high speed switching. When packets come into the switch, the cache is queried for the destination address. If the switch knows how to reach the destination, or the next hop towards the destination, then the packet is routed immediately. If no information is available about the destination, then a route discovery algorithm has to run to try learning how to reach that destination. Once the destination information is obtained, it is stored in a high speed cache, called Ternary Content Addressable Memory (TCAM), so that future packets going to the same destination are forwarded immediately.

1.1.1 Catalyst 6500 Series

For this project, the experiments are performed on a Cisco Catalyst 6500 series switch. The Catalyst 6500 switch offers high performance, and a feature-rich platform, suitable for deployment in campus, data center, WAN, and Metro Ethernet networks [18]. Some of the features provided by the 6500 series include: scale performance and network services, network virtualization, and network security[18].

The documentation on Cisco's website is rich and detailed, making it essential in determining the details for operating and configuring the switch. Some of the things that were investigated through Cisco's website included: configuring ftp, routing information protocol (RIP), and IP routing. The switch was configured with the bare minimum settings to make it learn destination IP addresses based on the incoming packets. On top of the documentation, Cisco Engineers have provided valuable implementation details regarding the workings of the TCAM, suggestions on resolving some of the issues involved with the experimental setup, and upgrading the software on the switch.

1.2 Motivation and Objectives

The work in this thesis will investigate how the performance of the switch is affected once the TCAM reaches its capacity. The purpose of the TCAM is to store prefixes of destination IP addresses, to shorten the time required to make a routing decision. The process of route discovery is relatively time consuming. In order to minimize the number of times the switch needs to figure out where a particular destination IP address is, it stores the prefixes of destination IP addresses in a high speed cache that can return queries in a very short amount of time. For the switch under investigation, once the capacity of the TCAM is reached, the switch starts doing Software Switching, which is significantly slower than Hardware Switching. If the switch has to constantly do software switching, then the throughput would be severely affected.

Initially it was thought that there is a replacement policy for the TCAM cache, which would replace entries in the TCAM once it has reached its capacity. Unfortunately, it turns out that there is no replacement algorithm for the TCAM, and instead the switch starts doing Software Switching instead of Hardware Switching. It is believed that the designers opted for increasing the TCAM capacity, as an alternative to dealing with a cache reaching its capacity. According to the Cisco engineers, the penalty associated with a replacement algorithm is too high, making the option of increasing the capacity of the TCAM a better option for maintaining high performance switching. Ideally, the minimum amount of TCAM should be used, because of the drawbacks of TCAMs: expensive, and high power consumption.

The first part of the project focuses on quantifying the penalty associated with a cache miss in the switch. This is important because the TCAM is one of the core components of the switch, and design decision regarding the amount of TCAM that should be used. Some of the design decisions that affect the cache are its capacity, and the algorithm involved in maintaining and updating the TCAM. It is important to know the exact penalty associated with a cache miss because it is considered a worst case scenario or the time required to do the packet forwarding/switching. The second part of the project focuses on quantifying the penalty associated with performing Software Switching, as oppose to Hardware Switching. This is again important because it tells the switch designers what the penalty associated with software switching is, and provide a way to benchark any future improvements of Software Switching.

The experimental setup created for this project could be useful for carrying out similar experiments as the one discussed in this thesis in a lab setting, enabling users to extract useful information.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 presents background information on Switches/Routers, Cisco's Catalyst 6500 E switch, cache memory, CAMs, TCAMs, and literature reviews of traffic characterization; Chapter 3 defines the framework for this project, the implementation details, the challenges that were present, and how they were resolved; Chapter 4 presents and analyzes the data collected from the experiments that were carried; Chapter 5 summarizes the findings of the thesis, and suggests some future work.

Chapter 2

Background

This chapter will discuss some of the background that is helpful for understanding the work done in this thesis. The main topics discussed are: switches and routers, Cisco's Catalyst 6500 series, cache memory, Content Address Memory, Ternary Content Addressable Memory, and literature review on Internet traffic characterization.

2.1 Switches and Routers

Switches and routers can be viewed as specialized types of computers. A computer is an electronic machine that is programmable, and can store and process data. A conventional computer consists of some type of memory, a module that can perform arithmetic and logic operations, and input/output devices to take in commands, and output results. Switches and routers have processing units, memory to store routing tables and operating system related information, and interfaces for accepting commands and outputting results - making them specialized computers.

A sample block diagram of a Cisco 1600 Series router is shown below:

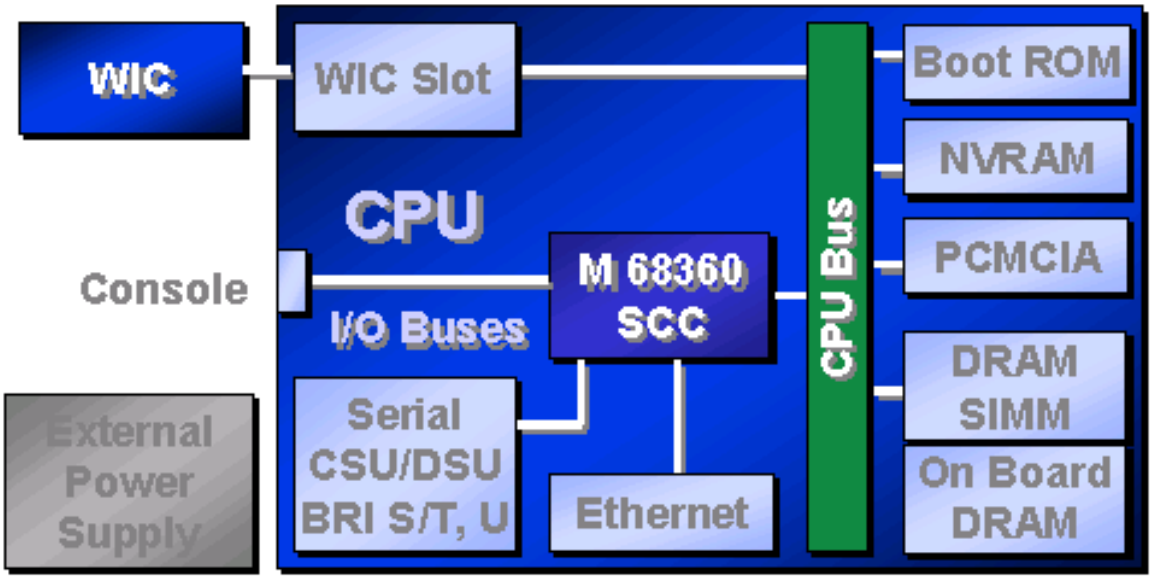

Figure 2.1: A simple block diagram of a router's internal components[17], showing a processing unit, memory storage, and input/output interfaces.

For this particular router, it has a module for performing arithmetic and logic operations, memory, and interfaces for communication purposes. The block labelled M 68360, a Motorola 68360 Complex Instruction Set Computer with a built-in Serial Communication Channels (SCC), is a module responsible for performing the arithmetic operations[17]. There are a number of different memory types on board, including Boot ROM and DRAM. Finally there is a console for entering commands and retrieving information from the router.

2.1.1 Cisco Catalyst 6500 E Series Switch

For the purposes of this thesis, the work was done on Cisco's Catalyst 6500 series switch. The specifications of the switch used to complete the thesis work are as follows:

Supervisor:	One sup720
Line Card:	One 48 GigabitEthernet line card
Operating System:	IOS Version 12.2(17R)
TCAM IPv4 Forwarding Capacity:	196,608 entries
Total TCAM Capacity:	1,048,576 entries
CPU:	SR7100 @ 600 MHz
Packet Buffer Memory:	8192 bytes

Table 2.1: A summary of the specifications of the Catalyst 6500 switch used for the work completed in this thesis.

The understanding of three concepts was required to complete this project: how the TCAM works and the information it stores, how the switch behaves when there is a miss and a hit in the TCAM, and how the supervisor engine works. The details of how the TCAM works are discussed in details in the next section of this chapter.

Determining how the switch behaves when there is a miss and a hit in the TCAM is essential for the understanding of the operation of the switch. The operation of the switch for Hardware Switching IP packets can be split into two cases:

- The switch has information about the destination IP address for the arriving packet
- The switch does not have information about the destination IP address, and has to perform route discovery

Examining the first case, where information about the destination IP address of the arriving packet is available, the following sequence of events occurs:

1. Extract the header information from the packet
2. Perform a lookup into the IPv4 cache
3. Perform a packet re-write - update the header information of the packet
4. Forward the packet on the appropriate interface - based on the information obtained in the second step

A diagram depicting the operation of the switch when information about the destination is available:

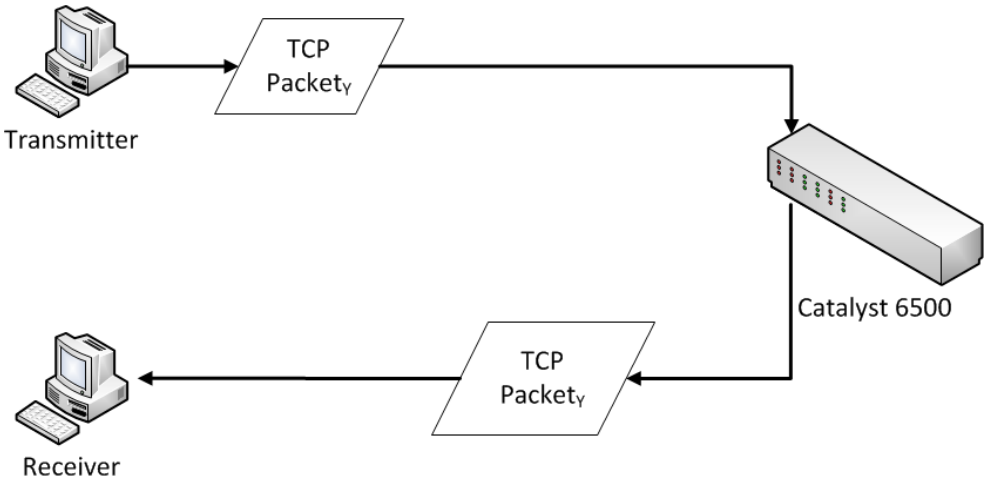

Figure 2.2: A diagram of how the switch behaves when the routing table of the switch has an entry for the destination of the packet. Note that the TCP packet is simply forwarded once the information of the packet's destination is retrieved from the cache.

Now we examine the second case for Hardware Switching, where information about the destination IP address of the arriving packet is not available in the cache:

1. Extract the header information from the packet:
2. Perform a lookup into the IPv4 cache
3. A miss occurs from the cache lookup. Place the packet in a buffer, and forward header information to the Supervisor
4. Forwarding engine in the Supervisor initiates the Address Resolution Protocol, generating an ARP request to get information about the destination
5. Assuming the destination replies to the ARP request, this information is forwarded to the Supervisor upon its arrival
6. Supervisor updates the IPv4 cache and performs a packet re-write on the buffered packet
7. Forward the packet on the appropriate interface - based on the information obtained in step 5

A diagram depicting the operation of the switch when information about the destination is not available:

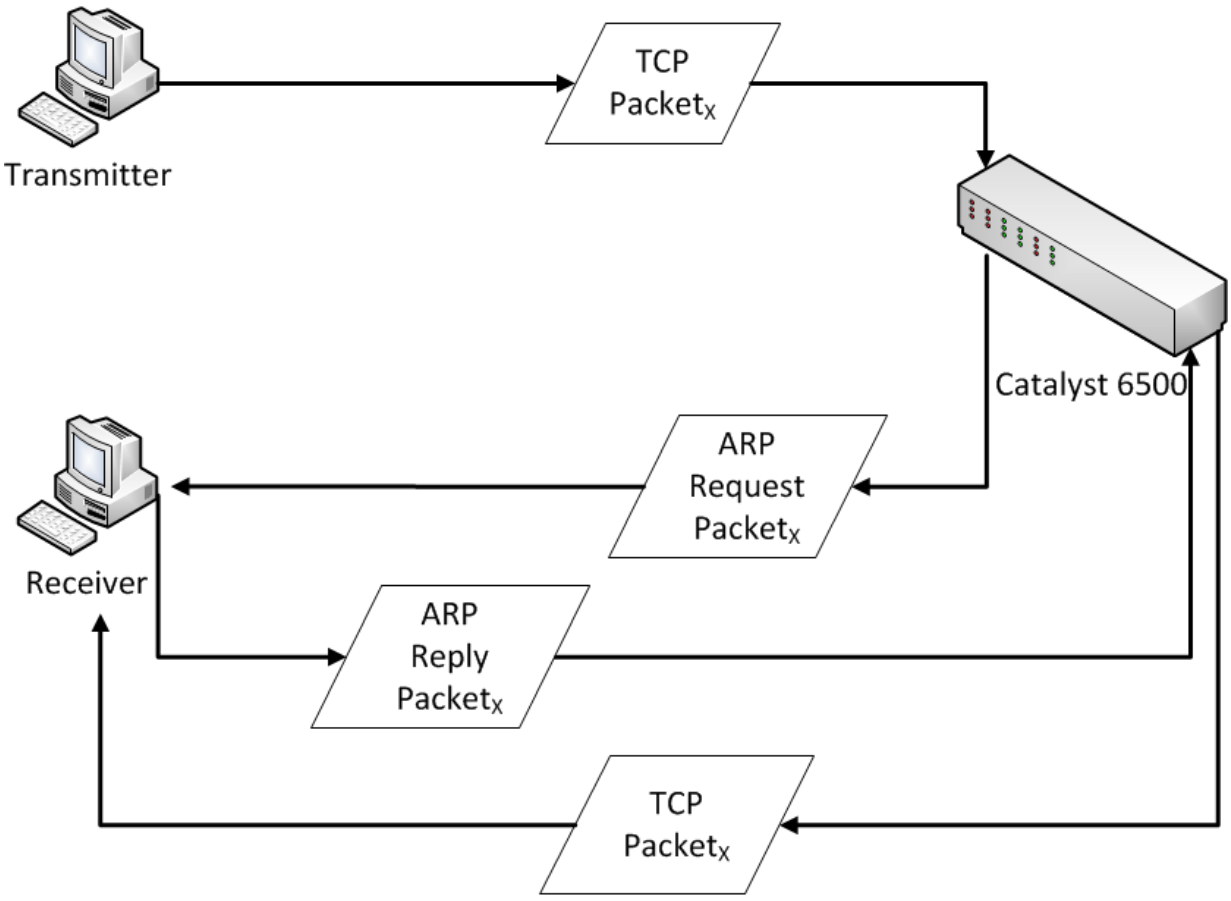

Figure 2.3: A diagram of how the switch behaves when the routing table of the switch does not have an entry for the destination of the arriving packet. Note that until the ARP reply comes back, the TCP packet does not get forwarded to the destination.

The behaviour shown in the figures above, figure 2.3 and figure 2.2, occur regardless if the switch is performing Hardware Switching or Software Switching - if there is a miss in the cache, it performs route discovery through an ARP request, and if there is a hit in the cache, then the TCP packet is forwarded. Details of how the supervisor is involved is discussed later on in this section, but the two main operational differences between Software Switching and Hardware Switching are:

1. Hardware Switching does not use the Supervisor when there is a hit in the cache on the line card
2. Software Switching always relies on the Supervisor, whether there is a cache hit or a cache miss in the cache, while Hardware Switching only relies on the Supervisor for cache misses

For demonstrational purposes, a diagram showing two arriving packets, one with a destination IP address in the cache, and one that does not have an destination IP address in the cache, is shown below:

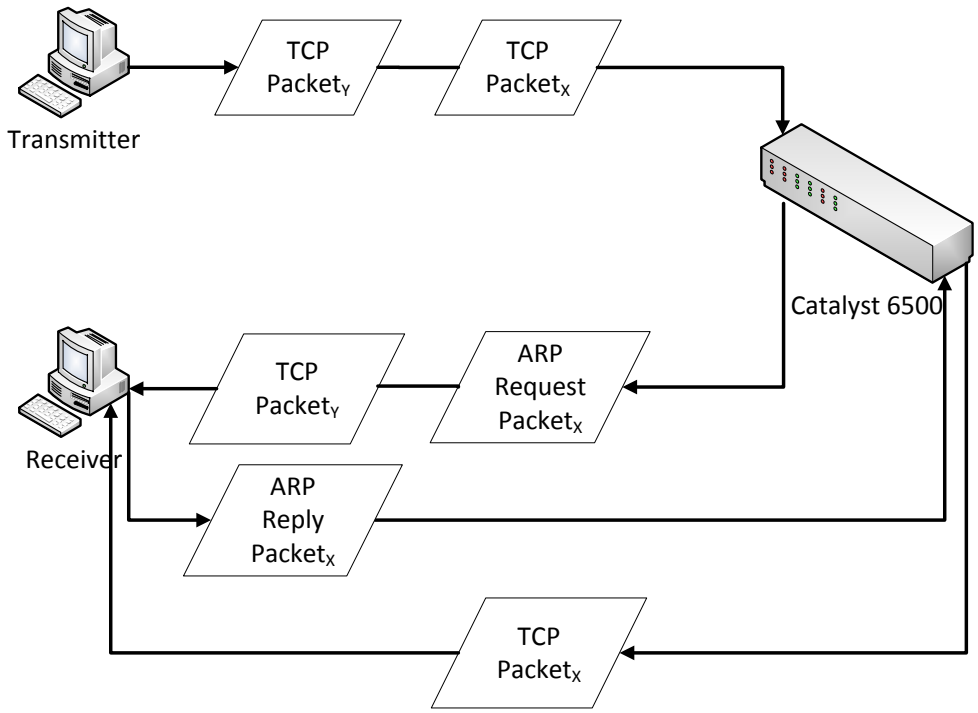

Figure 2.4: A diagram of how the switch behaves when the routing table of the switch has an entry for the destination of the packet, and when it does not. In this case, packet y has a destination IP address that is in the cache, while packet x does not.

Now that a general understanding of how the switch behaves has been established, we can talk about the Supervisor which contains a module called the Forwarding Engine. Two of the main responsibilities of the Forwarding Engine is to deal with packets that have a destination address that is not in the cache (Layer 2 and Layer 3 forwarding), and to perform Software Switching when the cache allocated for Hardware Switching has reached its capacity.

The Forwarding Engine uses the header information it receives to make a decision on the fate of the packet, while the TCP packet resides in a buffer. It first queries the cache that is allocated for Software Switching, if there is a hit, then a packet re-write is done and the TCP packet is sent out by the line card. If the there is a miss in the cache allocated for Software Switching, then it initiates Address Resolution Protocol. An ARP request is generated, and is sent out by the line card on the appropriate interface. The interface on which the ARP request goes out on is determined by the IP address and mask of the interface, where the interface must have a destination IP address and mask that would include the packet's destination IP address. For example, a packet destined for 100.8.0.10, could be sent on an interface with IP address 100.8.0.1/24, where the the range of addresses covered by that interface range from 100.8.0.1 to 100.8.0.255. If an ARP reply comes back for the address the switch is attempting to discover, then the Forwarding Engine is also responsible for processing it. When an ARP reply is passed on to the Forwarding Engine, the cache allocated for both software and hardware are updated with this information, and if the TCP packet is still in the buffer, then a packet re-write occurs and the TCP packet is sent out.

A high level diagram of how the forwarding engine works within the switch, for a cache miss, is shown below:

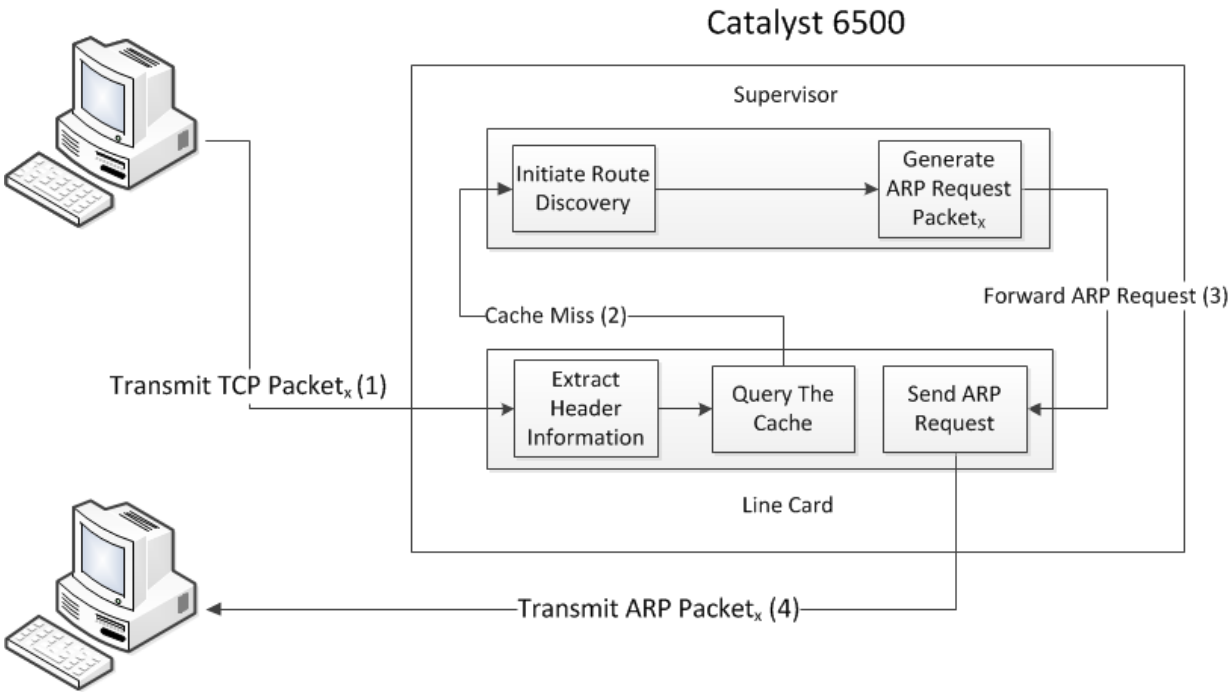

Figure 2.5: A diagram of how the Supervisor works within the switch, when a cache miss occurs. Note that in this case, the switch is attempting to do Hardware Forwarding, but a miss in the cache occurs.

As it was previously discussed, the Supervisor is not involved when Hardware Switching is being done.

Software Switching occurs when the cache allocated for Hardware Switching is filled up. As a packet arrives, the header information is extracted, and a TCAM lookup into the portion dedicated to Hardware is performed. Once a cache miss occurs, the information is for forwarded to the Supervisor, where the Forwarding Engine module performs a lookup into the cache allocated for Software Switching, and if the cache lookup is a hit, a packet re-write is performed and the TCP packet is forwarded. If there is a miss in the cache dedicated for Software Switching, an ARP request is generated and the TCP packet waits in the buffer until an ARP reply comes back.

A diagram showing the operation of the switch performing Software Switching, for a cache hit, is shown below:

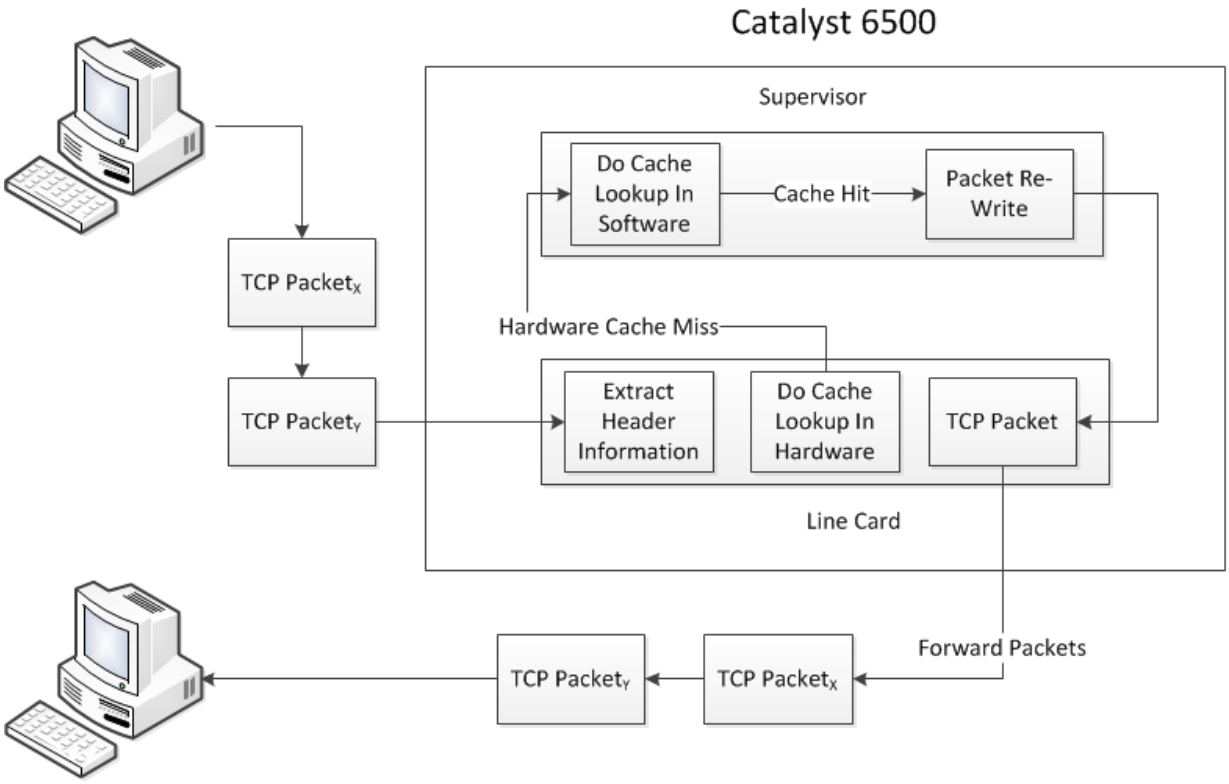

Figure 2.6: A diagram of how the Supervisor is incolved during Software Switching and information of the destination is available in the software cache. Note that in this case the packet information is directly sent to the Supervisor to deal with it.

Finally, a diagram showing the switch performing Software Switching, for a cache miss, is shown below:

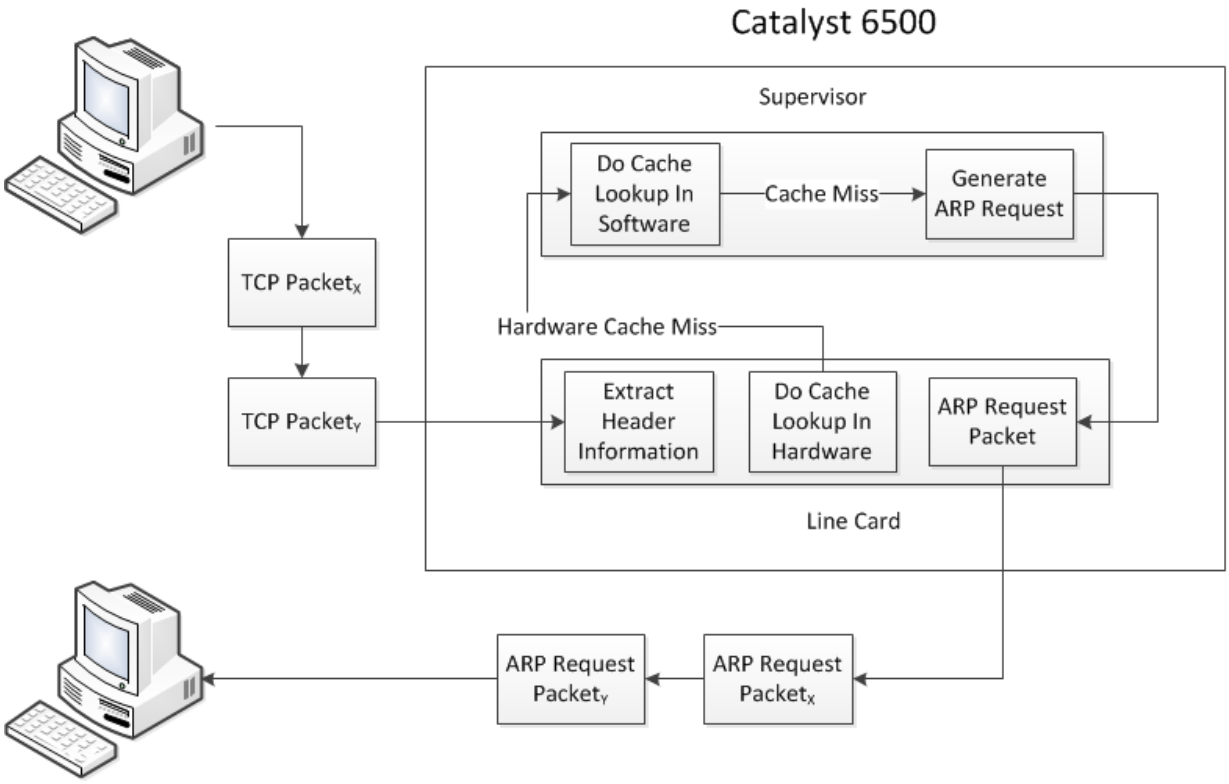

Figure 2.7: A diagram of how the Supervisor is involved during Software Switching, and the information of the destination is not available. Note that in this case the packet information is directly sent to the Supervisor to deal with it.

The time delay associated with a cache missed is calculated and discussed in Chapter 4. Software switching, and the time difference between forwarding packets using Software Switching and Hardware Switching is also discussed.

2.2 Cache Memory

Cache, is a type of computer memory, that is typically smaller in capacity, relatively expensive, and much faster than the ordinary memory[6]. The main purpose of cache

memory is to reduce the average access time of data for a system[4] . One of the many uses of cache memory is in modern day computer processors. The cache in processors is small, fast, and holds copies of data in the main memory - usually its the most recently used data to complete a set of instructions [5]. One of the reasons caches work well is because of a property of computer programs called the locality of reference, where most of their execution time is spent in routines in which many instructions are executed repeatedly during some time period [5]. The behaviour of locality of reference manifests itself in two ways: temporal, a recently executed instruction is likely to be executed again very soon, and spatial, instructions close to a recently executed instruction are also likely to be executed soon[5]. Through various replacement algorithms, blocks of data are copied from the main memory to the cache for manipulation, and then this data is copied back in to main memory once new data needs to be fetched into the cache. The cache is organized into cells, each storing one binary bit. The memory cells are grouped into words of fix word length, each accessible by a binary address of N bits. To read a value from a memory location, an address is supplied, and a value is returned. To write a value into a memory location, an address specifying the location has to be provided, as well as the value to be written into that location.

2.2.1 Content Addressable Memory (CAM)

Content Addressable Memory (CAM), is a also known as Associative Memory. "Associate memories have been generally described as a collection or assemblage of elements having data storage capacities, and which are accessed simultaneously and in parallel on the basis of data content rather that by specific address or location"[7]. This definition is for a specific type of Content Addressable Memory, one that not only accesses the memory by content, but also reads all the cells in parallel. Another more general definition given by Parahmi is: "Content Addressable Memory is a storage device that stores data in a number of cells. The cells can be accessed or loaded on the basis of their contents"[13]. CAMs differ from typical computer memory, which perform memory accesses by location - an address in memory is provided, and a value from that location is returned. In CAMs, when performing a read, you provide a value, and a get back a value associated with that memory location.

The implementation details and hardware operation of CAMs is highly complex, and beyond the scope this thesis. It should be noted that understanding these details is not required for the work done in this thesis.

2.2.2 CAMs in Catalyst 6500

One of the applications of CAMs is in high performance routers, known as Ternary Content Addressable Memory (TCAM). One of the important TCAM features the ability to include wildcard bits which will match both one and zero - beneficial for the use in IP lookup tables in routers and switches[1]. The main objective of the TCAMs in the router, is to store prefixes of learned routes in a routing table, to increase the forwarding speed for any future packets destined to the same location. Routers maintain a table of routing prefixes, each of which match a contiguous range of the Internet Protocol (IP) address space, and are associated with packet forwarding information[11]. For example, the router may have an entry in the cache with IP address 10.0.1.10, and a mask of 255.255.0.0 covering the range of IP addresses from 100.0.0.1 up to 100.0.255.255 - the mask being the determining factor, signifying the *don't care* bits of the address. When a packet arrives at a router, the destination address is extracted from the header information, and a lookup into the cache with this address occurs. The cache has to solve what is known as the Longest Prefix Matching Problem (LPMP)[11]. If the solution to this address exists, then it is unique and has the longest mask length of any other matching prefix. For example, assume the following simplified version of a TCAM holding a routing table in a switch:

Destination IP Address/Mask	**Outgoing Interface**
100.1.0.1/16	InterfaceGigabitEthernet1
100.2.0.1/24	InterfaceGigabitEthernet2
100.2.1.1/24	InterfaceGigabitEthernet3
100.3.0.10/32	InterfaceGigabitEthernet4

Table 2.2: Example of a simplified routing table stored in a TCAM cache.

If a packet, with destination IP address 100.2.0.10 arrives, then a lookup into the routing table will return InterfaceGigabitEthernet2. For address used in this example, the second (100.2.0.xx) and third entry (100.2.1.xx) match up to second most significant octet, but since the third octet in the destination IP address being quered matches with the third octet of the second entry in the table, the second entry is returned because it has the longest matching prefix. It should also be noted, than when the switch is storing information for Layer 3 forwarding, it stores the full destination address with a matching interface.

Some of the limitations of TCAMs are: high power consumption, high cost, and relatively small capacity[9]. The reason that TCAMs consume a lot of power is due to the large amount of comparison circuitry that is activated in parallel. A study to model TCAM

power consumption determined that a $0.18\mu m$ TCAM with $32k$ entries, each of length 36, requires $16.7nJ$ for a search access - compared to a SRAM of similar size requiring approximately $1.9nJ$ for a simple read operation[1]. The same study has also shown that doubling the number of entries of the cache, doubles the power consumption - a significant amount for a device performing hundreds of lookups every second. A lot of research has been done to optimize the use of TCAMs in order to reduce power consumption including: performing pre-computations to reduce number of TCAMs searched[12], taking advantage of some of the TCAMs' property to reduce static power by eliminating one of the subthreshold leakage paths[10], and a batch caching model to determine which routing prefixes to store[11].

The phrases cache, TCAM, and TCAM cache are used in this thesis from this point forward, all referring to the Ternary Content Addressable Memory in the Catalyst 6500.

2.3 Internet Traffic Characterization

As discussed in the Motivation section, the purpose of traffic characterization is to provide a reference point for the type of traffic the switch might have to deal with. This section will go through some previous publications on Internet Traffic Characterization, and then discuss the data characterized for this project. One of the main goals behind the literature review was to determine the defining factors for modelling traffic, and gaining insight on the different variables of the internet traffic. Another goal was to gain an understanding on the methodologies Internet traffic data is collected, how each one worked, and the advantages and disadvantages of each one. This was necessary for analyzing the data collected from the experiments carried out for this thesis work.

2.3.1 Previous Work on Internet Traffic Characterization

The main focus for the literature review was to investigate how Internet traffic characterization was performed, and find the most relevant method for the purposes of this research.

One of the publications reviewed, Wide-Area Internet Traffic Patterns and Characteristics, discusses a range of relevant topics including: the source of their data for Internet traffic characterization, traffic capturing implementation, and the methodology of Internet traffic characterization[21]. In this publication, the authors characterized their traffic over two time scales, 24 hours and 7 days - examining the traffic volume, flow volume, flow duration, packet size, and traffic composition. This publication served a few main purposes: a guide for the type of information relevant to characterization, the time span

over which the data is collected in order to get meaningful results, and the definition of Internet traffic flow. The authors defined traffic flow as: "A uni-directional traffic stream with a unique <source-IP-address, source-port, destination-IP-address, destination-port, IP-protocol> tuple"[21]. Some of their results for the data the authors collected on international links included:

	TCP Packets	UDP Packets
Percentage of Total Packets (%)	85-95	5-15
Average Flow Rate (packets/flow)	16-20	5-15
Average Flow Size (kilobytes/flow)	5-8	1-2
Average Packet Size (bytes/packet)	300	200-500
Average Flow Duration (s)	12-19	10-18

Table 2.3: Summary of some of the results from the paper "Wide-Area Internet Traffic Patterns and Characteristics (Extended Version)".

These results would have been used to form the type of Internet traffic for injection into the switch. All of the previously mentioned characteristics are controllable through the software generator used to complete the work in this thesis. Unfortunately the publication lacked the details of how to process the data to perform the characterization - they just presented their results. I was seeking more information on how to process the captured data to perform the characterization.

Another publication which was reviewed, Internet Traffic Measurement, discusses important concepts such as: measurement approaches (active versus passive), on-line versus off-line traffic analysis, and protocol level[22]. A passive network monitor is a non-intrusive device used to observe and record the packet traffic on an operational network, without injecting any traffic of its own onto the network[22]. "An active network measurement approach uses packets generated by a measurement device to probe the Internet and measure its characteristics"[22]. On-line traffic analyzers support real-time collection and analysis of network data, while off-line traffic analyzers perform analysis on collected and stored traffic. Finally different measurement tools collect and analyze data at different protocol levels. For the purposes of this thesis, data was captured and stored using the tcpdump utility, and off-line analysis was performed through a series of scripts written in Python - details in Chapter 3 and Chapter 4. The target protocol layer is Layer 3 (TCP/IP).

Another publication which was reviewed, Traffic flow measurement: Architecture, proved to be insightful for determining how the data is used to perform the characterization, and the details of a system which has been implemented to perform the process of capturing

and manipulation of the data[2]. In this publication, Internet traffic flow is defined as "a stream of packets passing across a network between two end points (or being sent from a single end point), which have been summarized by a traffic meter for analysis purposes"[2]. The author states that the definition of flow will depend on what you want it to be, including making the flow account for packets travelling in both direction (packets to destination and from destination are included in the same flow). You could even loosely define packet flow as a stream of packets having the same source and destination IP addresses[2].

2.3.2 Traffic Characterization of Data from CAIDA

In order to get more up-to-date information about current Internet Traffic characteristics, data collected by The Cooperative Association for Internet Data Analysis (CAIDA) was analyzed. The dataset used for this project is from the year 2010 called "The CAIDA Anonymized 2010 Internet Traces"[3]. For the characterization of this data, a flow is defined as a stream of packets sharing the same source and destination IP addresses, and the same source and destination TCP ports.

The dataset used contains anonymized passive traffic traces from CAIDA's equinix-chicago and equinix-sanjose monitors on high-speed Internet backbone links[3]. This data is useful for many researching characteristics of Internet traffic, including application breakdown, security events, geographic and topological distribution, and flow volume and duration[3]. This is particularly useful for us, because the distribution of traffic, flow volume, and duration, are all factors that directly affect the cache under investigation, because it affects things like: the data being stored in the cache, how the data changes, and the frequency of change. For obvious security reasons, the traffic traces are anonymized using CryptoPAn prefix-preserving anonymization, and the payload from all the packets has been removed [3]. This does not affect our work because the payload itself does not affect our results. It should be noted that while the publishers claim that the network cards used to record the traces provide timestamps with nanosecond precision, the pcap files provided have truncated precision to microseconds [3]. We too, are also using pcap files to store and analyze files.

Chapter 3

Framework

There are two major components for this project, the software developed for analyzing the collected data, and the setup of the environment for conducting the experiments - which is compromised of hardware and software.

3.1 Internet Traffic Characterization

Python was used to automate the process of extracting and analysing Internet traffic information collected by CAIDA. For the purposes of extracting data from Internet traffic, a script was written to extract the necessary information from '.pcap' files, and then organize and group that data in a certain way to collect meaningful characteristics.

By the definition provided in the background section, a flow is a stream of packets where the source and destination IP addresses, as well as the source and destination TCP ports, of packets are the same. The fields discussed in the definition of flow are extracted from the capture file to correctly identify the flow a particular packets belongs to. The details of how the data is organized and characterized in discussed in the subsection Extraction of Useful Data. The characterization that was done on the data is: the number of packets per flow (flow size), the average inter-packet delay per flow, and the flow duration.

The flow size was straight forward to calculate: when a packet that belonged to a particular flow was encountered in the data being analyzed, a global counter keeping track of the flow size for that particular flow was incremented. For the flow duration, a global counter would sum the inter-packet delays of packets belonging to the same flow as they

are encountered. The inter-packet delay, t_x, for flow x, flow size y, and inter-packet delay for packet i, t_{x_i} can be represented using the equation:

$$t_x = \sum_{i=0}^{i=y} t_{x_i} \tag{3.1}$$

For the flow average inter-packet delay, t_{AVG_x}, an array variable for each flow kept track of the inter-packet delay for each packet in that flow, and once all the data has been parsed, a calculation would be done to determine the average by summing the inter-packet delays, t_x, in a flow and dividing it by the flow size, y.

$$t_{AVG_x} = t_x/y \tag{3.2}$$

3.1.1 Extraction of Data From CAIDA Dataset

The data captured from CAIDA is split into a large number of files, and grouped by approximately one minute of capture time. The capture files have the ".pcap" extension, and contain a large amount of information about each individual packet that has been captured. Even though the source and destination IP addresses have been made anonymous, it has been done in such a way so as to preserve the relation between the source and destination and addresses. That means, even though the IP addresses are not real, they can still be used as a substitute. For the purposes of this project, the information that useful, and necessary for the characterization of the Internet traffic, are shown below:

- Source and Destination IP address
- Source and Destination TCP ports
- TCP time delta, which is the time since the previous frame in this TCP stream has been received[24]
- The packet payload size

The first two items listed above, source and destination IP address and TCP ports, were required to sort a packet into the right flow. The third item was used for calculating the inter-packet delay, and flow duration. Finally the payload was used to calculate the flow size. The extraction and manipulation of the data from the tcpdump files was performed

using custom scripts created in Python. The code for the data extraction is shown in the Appendix A, section A3 .

The first step in the extraction process is to run a tshark command which requests specific information as discussed earlier. The tshark command returns tuples of data based on the provided filter included in the command. In this case, we are only interested in seeing the source and destination IP address, source and destination TCP ports, frame relative time, and the length of the stored TCP packets. Once this data is stored in a variable, I loop through it to build a dictionary of the Internet packet flows. The alternative to having a dictionary is storing the information in an array of flow. The problem with having an array of flows is it will require a nested loop every time we need to find particular flow to sort the packets. This would increase the runtime complexity significantly, and slowdown the parsing of the data as the number of processed packets increases. Once we finish processing all the packets and build our dictionary, we loop through to calculate dictionary to calculate the flow related information: flow rate and duration. Finally, the results are written to a comma separated file for plotting. Details of the implementation are discussed in the next chapter.

3.2 Experimental Setup

The setup for this project had to be able to send and receive packets, allowing the switch to perform Layer 3 forwarding. The setup used for this project involves two computers and one Cisco Catalyst 6500 switch. A diagram of the setup is show below:

3.2.1 Requirements

A setup had to meet a number of requirements in order for us to successfully complete the work of this thesis:

- The ability to fill up the TCAM cache
- The ability to view the switch's cache utilization and content
- The ability to generate packets at a high rate
- The ability to generate custom packets

Since the project revolves around analysing the effects of over flowing the TCAM cache, one of the main goals of the setup was having the ability to fill up the cache. The setup had to be flexible enough such that you can fill up the cache at varying rates, because depending on the experiment being carried out, the requirements for how fast the cache has to be filled up can be different. For example, if we are measuring the cache miss penalty, the transmission rate requirements would be different than when trying to measure the amount the cache hit performance, where the later must be as fast as possible. An alternative to being able to fill up a large cache would be having the ability to re-size the cache, and performing any necessary work on that instead. Re-sizing the cache would shorten the time of an experiment, allowing for more experiments to be carried out in a shorter time frame. It was also important to be able to view the cache size, utilization and contents to make sure that the setup was working correctly.

The third key requirement, crucial for this project, is the ability to generate packets at a very high rate. The reason the packet rate had to be large is because it was necessary to make sure that the inter-packet delay calculated at the receiving end, was entirely due to the speed at which the switch is forwarding the packets at, and that the transmission rate had little to no effect on the inter-packet delay - something that could only be achieved by pushing the switch with high transmission rates.

The fourth requirement was the ability to generate custom packets. This was critical because on the transmitter side, we wanted to generate TCP/IP packets, with certain source and destination IP addresses, source and destination TCP ports, and a custom payload. On the other hand, on the receiving side, we wanted to be able to generate ARP packets. Without this, we would not be able to fill up the cache.

3.2.2 Challenges

There were a number of challenges in order to get the setup working:

- Figuring out how the switch works
  - How can we configure the switch to complete our work?
  - How can we monitor the cache and retrieve useful information about it?
  - What does the TCAM cache actually store?
  - How do entries get added to the TCAM cache?
- Figuring out how to deal with ARP requests being sent by the switch.

- Figuring out how to generate custom made packets at a high rate through software
- Figuring out which devices should be utilized for the experiments

The first challenge was figuring out how the switch exactly works, and the role that the TCAM plays in switching. This was especially challenging because I had no previous working experience with the Catalyst 6500. One of the key requirements for the project was to configure the switch to learn routes, and dynamically update its cache appropriately - otherwise filling up the TCAM would be difficult. The switch has a number of IP routing protocols such as: static routing, Routing Information Protocol (RIP), and Enhanced Interior Gateway Routing Protocol (EIGRP)[19]. Each protocol was examined, to determine the relevant implementation details, and see the advantages and disadvantage of each one. The criteria for the routing protocol was: easy to implement and monitor, dynamically adds routes to the TCAM, and performs at a speed that can keep up with the packet rate of incoming traffic.

The next step was to figure out how to view information about the cache. Some of the necessary information included: the size of the cache, how much is currently in use, and the contents of the cache. Knowing the size of the cache is required to determine how many destination IP addresses are needed to fill it up. Second, being able to monitor how much of the cache is in use and its content are necessary to make sure that experiment was running correctly. For example, if a packet is sent to address 100.8.0.5, and the switch does not know where that ip address is, then it will learn the route and add an entry into the cache with the appropriate destination interface. So printing the contents of the cache will show whether the setup is working correctly or not by showing a new entry in the cache, or lack thereof. Once we are able to view the contents of the cache, that will give us insight into what is exactly being stored, and help determine what would fill up the cache.

After figuring out and verifying the workings of RIP and monitoring the cache as the packets are being fed into the switch, the next step was to figure out how we will deal with the ARP requests being sent for the destinations that are unknown to the switch. Since the switch will only forward packets to the second computer if there it has information about the destination in the cache, or if it receives an ARP reply from the destination being sought, we had to figure out a way to make the computer reply to ARP requests as they come in to the receiver. In other words, the goal was to turn the second computer into a traffic sink. The first approach investigated was to utilize the Linux operating system to automatically do this. The idea was to make use of some of the networking features already implemented in the the operating system, but I was unable to do that. It was then concluded that the only solution was to listen on the Ethernet interface and when an ARP

request is received, generate an appropriate ARP reply and transmit it back to the switch. While this would solve the problem of replying to ARP requests, it introduced another challenge: how to generate the ARP replies fast enough to keep up with the transmission rate of packets being fed into the switch. This was difficult because it involved three steps: listening on the Ethernet interface for ARP requests, extracting the destination address being queried by the switch to craft an appropriate ARP reply, and finally transmitting the ARP reply. Since this will be done in software, it was going to be relatively slow. Part of the solution to this problem was also tied to the next challenge - the ability to generate custom made packets at a high rate.

There were two alternatives available to generate custom made packets: use an application that takes in a set of parameters and have it generate the desired packets, or write a program that will generate and transmit packets. The problem with the first approach is the lack of availability of good software that can do this kind of thing. A program called Nemesis was found, which is capable of generating packets based on certain parameters such as source and destination IP addresses, source and destination TCP ports, and a given payload file. The problem with Nemesis was that when the command to generate the packets was scripted, there were frequent errors and malformed packets. More details about overcoming this challenge is discussed in the next section. It should be noted, that it would have been very difficult to complete this project, without being able to generate packets at a very high rate - the slower the packet generation and transmission process was, the harder it was to pick up any meaningful results from the captured files. If it takes time t to process packets that are in the cache, then it take $t + p$ to process packets that are not in the cache - where p is a penalty associated with a miss in the cache. If the packet transmission rate is not high enough, then the penalty associated with a cache miss would not be detectable at the output. The delay between two packets being transmitted has to be less than or equal to p. A sample of the large difference between the inter-packet delay associated with routing packets is shown below:

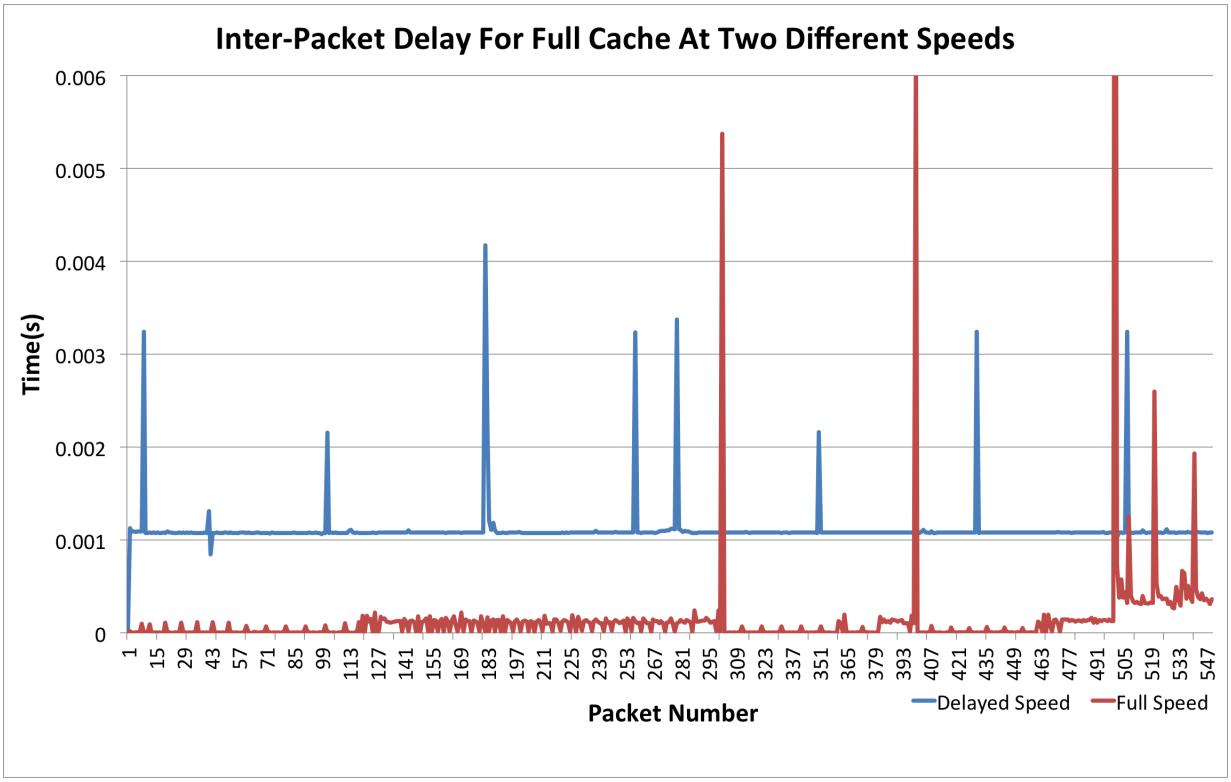

Figure 3.1: Inter-packet delay for varying transmission rates. Note that the increase in transmission rate, reduces the inter-packet delay, confirming the direct effect of the transmission rate on the inter-packet delay.

The blue line shows the inter-packet delay associated with a low packet transmission rate - approximately in the range of 60 to 70 packets per second, while the red line shows the inter-packet delay associated with a high packet transmission rate - approximately in the range of 60,000 to 65,000 packets per second. The difference in time between the two scenarios, on average, varies by more approximately a factor of 7. Details summarizing some of the statics of the inter-packet delay on the different transmission rates is shown below:

Calculation	Inter-Packet Delay LR (s)	Inter-Packet Delay HR (s)
Max Value	4.17×10^{-3}	13.40×10^{-3}
Min Value	844×10^{-6}	1.00×10^{-6}
Average	1.10×10^{-3}	141×10^{-6}
Standard Deviation	233×10^{-6}	689×10^{-6}
Variance	5.43×10^{-8}	4.75×10^{-7}

Table 3.1: Inter-packet Delay statistics for a low rate transmission rate, LR, and a high transmission rate HR.

We can see from the data above that as the transmission speed increases, the inter-packet delay drops significantly. The idea is to find the point in time in which increasing the transmission rate, will not decrease the inter-packet delay time. Reaching that point is essential to improving the results collected during the experiments.

The last phase of the project was to figure out which devices to use. There were three Catalyst 6500 switches, and two 7200 routers available for use. The 7200 routers have high bandwidth, and they would've been ideal for the experiments performed but unfortunately, the software running on the router was not flexible enough to be used as a packet generator because there was no way to generate custom made TCP/IP packets. The only thing that was possible was to generate the same packet repeatedly. It was also not possible to utilize the router as a traffic sink, therefore the decision was made to exclude it from the setup. Finally, a decision had to be made as to how many switches should be used in the setup. We have three identical Catalyst 6500 switches in the lab that could be used and it was determined that for the purposes of this project, the effect of the penalty would most likely be aggregated over the two or three switches, which would not reveal any additional information about the cache penalty. Thus, the final setup included two computers, a packet generator and a traffic sink, and one Catalyst 6500 switch.

3.2.3 Overcoming the Challenges and Meeting the Requirements

Due to the lack of experience configuring, and using the CAT6K, the first challenge was to figure out how to configure the switch. To overcome this challenge, we had to figure out what we wanted the switch to do. We knew that we needed a dynamic IP routing protocol capable of learning routes and adding the relevant information into the TCAM. This challenge was overcome by examining Cisco System's on-line documentation. The various IP routing protocols are discussed briefly in the referenced link, and more details were obtained by following the relevant links on the page or by looking up each protocol individually[19]. After examining the available documentation, Routing Information Protocol (RIP) was chosen because it was simple to implement, for detecting the devices directly connected to the switch. The configuration details are included in Appendix B, section B1. Routing Information Protocol (RIP) is a distance vector routing protocol, which uses broadcast User Datagram Protocol (UDP) data packets to exchange routing information - every 30 seconds in Cisco's IOS software[20]. If a router does not receive an update from another router (or connected device) for 180 seconds or more, it marks the routes served by the non-updating router (or device) as being unusable and removes the relevant entry from the routing table if there is still no update after 240 seconds[20]. It should be noted that RIP is not the preferred choice for routing as its time to converge and scalability are poor compared to other protocols such as: EIGRP, OSPF, or IS-IS[8]. The reason that the convergence and scalability of the protocol was not an issue for us, is because the computers involved were only one hop away from the switch, since they were directly connected.

The next crucial step in the project was figuring out how to generate custom made packets at a high transmission rate. One of the early options examined was finding software which ran on Linux, capable of generating packets at a high rate.

The first software which was stumbled on was Nemesis[15]. As advertised on the software's website, its a utility well suited for testing Network Intrusion Detection Systems, firewalls, IP stacks, and a variety of other tasks[15]. As a command-line driven utility, Nemesis is perfect for automation and scripting[15]. One of the main advantages of this utility is that it was a command line tool that was very simple to use. For example, to create a TCP packet, all you had to do was open a terminal and run the following command: sudo nemesis tcp -v -S 205.153.60.236 -D 205.153.63.30 -P payload.txt x 20 y 40. The options in the command are to define the parameters of the packet being sent, where -S specifies the source IP address and -D specifies the destination IP address. The previous example demonstrates the level of flexibility available with this utility, where the user is able to provide a great deal of parameters for the packet they are interested in

creating. The utility is capable of creating a wide range of packet types including ARP, DNS, ETHERNET, ICMP, IP, RIP, UDP and IGMP. Allowing the user to create ARP packets as well, was necessary for responding to the ARP requests on the receiver end. In order to make the use of this utility feasible for this project, a Python script was written to generate nemesis commands with the appropriate fields in order to transmit packet. Unfortunately, upon performing a few tests with different sets of data, it was concluded that this utility would not work. The first issue was that it had slow packet injection rate. The scripted commands could only achieve a transmission rate ranging from 8 to 17 *packets/s*, which was inadequate. The other issue was that while the script is running, there would be batches of packets that fail to inject, with the application reporting a meaningless error message "error injecting packet". Unfortunately even in debugging mode, it was unclear why some of the packets were not being injected correctly. Some of the packets that were faulty during the scripted packet injections, worked correctly when it was done manually, and even re-running some of the scripted packet injections caused them to work correctly - making the problem even more mysterious.

Many attempts were made to find a utility that was similar to Nemesis in flexibility and ease of use, but unfortunately they were all futile.

The next solution that was stumbled on was Scapy, an interactive packet manipulation program[16]. Scapy can run as an independent application, or it can run under a Python script. To test its flexibility, and make sure that it would work well for the intended purposes, Scapy was started as an independent application, and a few tests were carried out. Using Scapy was again very simple and highly flexible. For example to create a TCP packet with source IP address 1.2.3.4, destination IP address, source TCP port 20, and destination TCP port 40, you would use the following line of code: pkt = Ether()/IP(src='1.2.3.4', dst='129.0.0.1')/TCP(sport=20, dport=40). Note that leaving one of the layers empty meant that Scapy should use the defaults for that layer. After the packet is created, the send command would push the packet out on a specified interface using the following command: send(pkt, iface="eth0").

After making the decision of using Scapy, the first problem that was stumbled upon was the ability to achieve high packet transmission rate. Initially, the script written would loop through and do the following:

- Generate a destination IP address using counters that keep track of each octet of the address
- Create the custom IP packet with the destination address generated in the previous step

- Send the packet
- Update the counters for the octet

Unfortunately this approach was very slow due to the number of nested loops that were in the script performing the address updates. That is, we would iterate through the range 1 to 252 for the fourth octet, then go to the outer loop and update the third octet (which also ranged from 0 to 255), and finally we would go to the outer loop to update the second octet (which ranged from 8 to 30). This achieved a packet transmission rate in the range of 10 to 20 packets per second - an improvement over Nemesis, but still inadequate.

To further improve the previous implementation, the generation of IP destination addresses were removed from the script, and done independently in a different script. Since the destination addresses were not going change between different experiment runs, using a different script to permanently generate the destination addresses and write them to a file made sense. This removed the redundancy of generating the destination IP addresses every time we wanted to run an experiment, and speed up the run time of the script. The original script was changed to read the text file containing the destination IP addresses, and store it in an array for use. Using one loop iterating through the destination IP addresses array, packets were generated and sent out. This was a large improvement because it removed the nested loops and replaced them with just one loop. This change in implementation achieved a packet transmission rate in the range of 40 to 50 *packets per second* - a rate still considered to be slow.

A small improvement on the previous implementation that was attempted, was to run multiple instances of the same script, but transmitting packets in different destination IP addresses ranges. This worked well for up to three instances running in parallel, reaching a packet transmission rate in the range of 100 to 150 *packets per second*. The problem with this solution was it put a lot of strain on the computer's processor and memory resources, something that was visible through the computer monitoring tools. The other issue with this approach was that it had a diminishing return after the third instance. Meaning, running four or five instances in parallel had minimal improvements over the three scripts in parallel scenario. Therefore, running more instances in parallel was an improvement, but still fairly limited.

After doing some more research about improving the transmission rate, I found out that Scapy's built-in functions for creating and transmitting packets were indeed limited in terms of speed. Therefore the next iteration of improvement was to use Scapy to create a socket and send the packets on it. In order to do that, you had to assemble the packet, and then 'build' it before transmitting it. The build function takes the parameters provided

for the packet being built, and converts it to machine language - something that resembles an actual packet travelling on the network. This solution was a vast improvement in packet transmission rate over anything previously discussed - reaching transmission rates in the range of 160,000 to 170,000 *packets per second*. The previously mentioned rate was only achieved if you sent the same packet in a loop. In our case, we wanted to send different packets, which could only reach a rate of 60,000 to 75,000 *packets per second* - an acceptable rate. The downside of this implementation is the fact that building the packets was extremely time consuming, and had to be done pre-transmission, otherwise the transmission rate is significantly reduced. To overcome this issue, a loop at the start of the script would create the packets, build them, and store them in an array. Next, a loop would go through and transmit the packets. This solution was not ideal because the script had to run for a long time before the transmission began. To overcome this issue, and to avoid the redundancy associated with building the packets every time we wanted to carry out an experiment, a script was written to build the packets and write them to a file. Then another script would simply read the file containing the pre-built packets, and iteratively transmit them. To speed up the response to ARP requests on the receiver side, a similar approach was used to build and store the packets.

Implementation Improvement	Transmission Rate (packets/s)
Pre-generate destination IP addresses and write to file	40 to 50
Run multiple instances of the script in parallel	100 to 150
Pre-build packets, and transmit through the socket directly	60,000 to 75,000

Table 3.2: Summary of the improvements carried out to the original script and its effects on the packet transmission rate.

To verify the transmission rates, packets that were being transmitted on the Computer1 were recorded in a capture file and later on analyzed using the program *capinfos* to get information such as the number of packets captured, the average data rate in bytes/s and the average packet size in bytes. This information was used to verify that all the packets that should be transmitted, have in fact been transmitted. Using the average packet size and the average data rate, we were able to extract the average packet rate. The average transmission rate (*packets/s*), TR_{AVG}, is equal to the average data bit rate (*bytes/s*), BR_{AVG}, divided by the average packet size (*bytes/pkt*), PS_{AVG}. The equation to perform this calculation is shown below:

$$TR_{AVG} = BR_{AVG}/PS_{AVG} \tag{3.3}$$

A few samples of the output of capinfos, showing the information used to determine the transmission rate, are shown below:

Capinfos Output
File name: experiment1.pcap
Number of packets: 3560
Data byte rate: 3191693.44 *bytes per sec*
Average packet size: 54.29 *bytes*

Table 3.3: Sample output of Capyinfos. This data was used to ensure that all the packets were successfully transmitted, as well as determine the transmission rate in packets per second.

Capinfos Output
File name: experiment2.pcap
Number of packets: 3560
Data byte rate: 3441051.05 *bytes per sec*
Average packet size: 54.29 *bytes*

Table 3.4: Another sample output of Capyinfos. This data was used to ensure that all the packets were successfully transmitted, as well as determine the transmission rate in packets per second.

The next step was figuring out how to fill up the TCAM. One of the early approaches investigated was to assign a large number of IP addresses to the switch's interfaces, and then use *policy based routing* to have the packets forwarded from Computer 1 to Computer 2. The assignment of the IP addresses to the interfaces served the purpose of filling up the cache, while the *policy based routing* made sure that the packets were forwarded to the receiving computer. The problem with this approach was that the cache capacity is indeed large, and so a large number of IP addresses were required to fill it up. The second issue associated with this approach was that the switch became extremely slowly because RIP had to transmit the information of all these interfaces to the connected computers every 30 seconds.

Finally the approach that was taken involved figuring our how to reply to ARP requests being sent out by the switch. Since the switch will only forward packets to the second computer if it receives an ARP reply for the destination address of the packet its trying to forward, we had to figure out a way to make the computer reply to ARP requests as

they come in. In other words, the goal was to turn the second computer into a traffic sink. Making the switch automatically answer ARP replies by changing the operating system's network settings related to ARP, were unsuccessful. It was then concluded that the only solution was to listen on the Ethernet interface and when an ARP request is received, generate an appropriate ARP reply and transmit it back to the switch. While this would solve the problem of replying to ARP requests, it introduced another challenge: how to generate the ARP replies fast enough to keep up with the transmission rate of packets being fed into the switch. There were a few problems related to this issue:

- How to actively listen on the Ethernet interface for ARP requests
- How to extract the relevant information from the ARP request
- How to reply to a high volume of ARP requests

There were a few approaches considered to overcoming these challenges. First, it was realized that we can take advantage of the fact that we know the destination IP addresses of the packets being transmitted. One approach that was tested out, was to pre-emptively transmit gracious ARP messages. The purpose of gracious ARP messages is to announce your IP address and your hardware address, to the devices you are directly connected to. This would initially reduce the amount of ARP requests the receiving computer has to deal with as a result of the packets being sent into the switch. Unfortunately, that approach did not work because the switch did not add entries into the TCAM based on gracious ARP packets - it had to receive a TCP packet destined to an unknown location, and then process an ARP reply to add an entry into the TCAM. Another, closely related approach that was initially considered was to just send out ARP replies when the transmission of IP packets started, without having to first listen to an ARP request and then craft an appropriate reply. Since the order of packets, and the destinations were known, the script transmitting ARP reply packets would be started at the same time as the script transmitting IP packets on the transmitter computer. This did not work either because it was not possible to get the two computers in sync, causing a large number of packets to be dropped. Therefore I reverted back to the approach of listening on the Ethernet interface for ARP requests and send out the appropriate ARP reply.

The Scapy library, in conjunction with Python, were used to listen on the interface for ARP requests, extract the necessary information, generate an ARP reply, and finally transmit the ARP reply. The focus then shifted towards speeding up this process. The thought process behind speeding up the response of the receiving computer involved examining the following:

- What type of pre-processing can be done to help improve the performance?
- Are there any software concepts that can be utilized to improve the performance of the script?

The initial script for the receiving computer first pre-built ARP packets for the destination IP addresses of the incoming packets, and then stored them in a list. The script would then listen on an interface for ARP requests, extract the destination IP address, and loop through the list to find the appropriate pre-built ARP reply. This approach had two issues: the most time consuming part of this process was the extraction of the destination IP address from the ARP request, and the generation of the ARP reply packet; secondly every time you searched for a pre-built packet, the run time was *O(n)* - where n is the size of the list. While it did take a bit of time to build an ARP packet, it was much faster than a TCP packet. The initial script would listen on the Ethernet port, extract the destination IP address from the ARP request. It would then generate an ARP reply with the source hardware address as the receiver computer's Ethernet hardware address, the destination hardware address as the switch's Interface, and the destination IP address as the switch's interface. The next step was to transmit the created ARP reply. Unfortunately, it was clear from the number of packets that were being dropped that this script could not keep up with the switch.

The next improvement that was explored for this script was to pre-build the ARP replies and store them in a list - similar to the approach used for TCP packets. Again, this approach was trying to take advantage of the fact that we knew the destination address of all the packets that were going to be transmitted. The script would start off by reading a file with all the destination IP addresses, pre-build the ARP replies, and store the packets in an array. Once an ARP request was received, again the destination address was extracted, and a loop would iterate through the list of built packets and transmit the matching one. While this improvement allowed for slightly higher transmission rates, it was still not ideal.

Finally, the solution that seemed to work best was the use of the *dictionary* data structure. The main advantage of this data structure is that it offers approximately *O(1)* run time for searching the pre-built packets - a major improvement over the run time of searching through a list, which is *O(n)*, n being the size of the list. The final script for ARP replies is included in Appendix A, section A2.

3.2.4 Devices' Connections and Configuration

The first device used was a standard dual core computer, with a processor clocked at $3GHz$, $1GB$ of RAM, and running Ubuntu 10.04 (Kernel 2.6.32-24-generic). The computer was used to generate packets and send them to the switch using a Python script which utilizes the Scapy library. The scripts for performing the packet generation and transmission are included in the Appendix A, section A2. In the original setup, three Ethernet cards were installed and connected to the switch. When the data throughput being generated was too low, using the three Ethernet cards to feed packets into the switch was the first thing that was tried out. Unfortunately, as it was previously discussed, this did not yield any significant improvements in terms of the throughput of the packet generation. Therefore, Ethernet card 0 was the only one used, connecting it to interface *GigabitEthernet8/1* on the Catalyst 6500. The network configuration file is included in Appendix B, section B2.

The routing table on the computer had to be updated by adding entries into the network configuration file, in order to allow the operating system to determine which interface to send the packets out on. For example, if we wanted the packets with destination addresses in the range of 100.8.0.1 to 100.8.255.255 to go out on interface *eth0*, then we would add the following command to the network configuration file (/etc/network/interfaces): *route add -net 100.8.0.0 netmask 255.255.0.0 eth0*. Note that the interface had to be restarted after any change were made to this file, through the following command *sudo /etc/init.d/networking restart*.

The second device used was a CAT6K. It was configured for basic Layer 3 forwarding. In the original setup, three interfaces were used to for feeding packets into the switch from Computer 1, and three interfaces were used for pushing out packets to Computer 2. In the final setup, only one Ethernet card on the transmitting and receiving end were used, thus only two of the interfaces were in use (interface *GigabitEthernet8/1* and *GigabitEthernet8/25*) - one for receiving packets and one for transmitting packets out. The switch was configured to have a few addresses assigned to interface connected to the receiver computer to accept a large range of IP packets. The configuration file used for the switch is included in Appendix B, section B1.

The third device used was another dual core computer (core 2 duo), with a processor clocked at $2.33GHz$, $2GB$ of RAM, and running Ubuntu 10.04 (Kernel 2.6.32-24-generic). The second computer was used to listen for ARP requests and reply with the appropriate ARP reply packet. The process of listening and replying to ARP packets was also performed using a Python script utilizes the Scapy library. The Python scripts used to perform these tasks are included in the Appendix A, section A2. As previously discussed, since the original setup involved having three Ethernet cards for transmitting packets, there were

also three Ethernet cards on the receiving end capturing the packets. In the final setup though, only one Ethernet card was used, Ethernet 0 and it was connected to interface *GigabitEthernet8/25*. The network configuration file of computer 2 is included in Appendix BB, section B3.

A high-level representation of the setup used for this project is shown below:

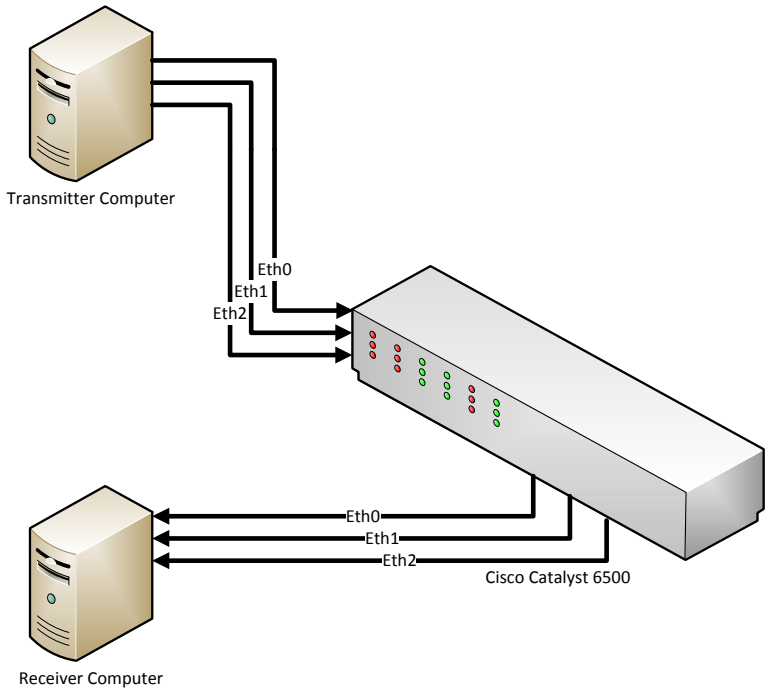

Figure 3.2: This is a high-level representation of the setup used for the work completed in this thesis. The setup includes two computers and a Cisco Catalyst 6500 switch.

3.2.5 Performing An Experiment

In order to perform an experiment, a few things had to be performed before the experiment began. First, the packets that are to be used for the experiment had to be generated. The next step was to start the script on the receiving computer to have all the ARP packets ready to start replying right away as the packets came in. Once the script on the receiving end had all the ARP reply packets, the capture application on the receiving computer was started. Next, the script for transmitting the packets is started, and once it finishes running, the capture file is stopped.

A summary of the steps is shown below:

1. On the transmitting computer, pre-build the TCP packets that are going to be used in the experiment
2. On the receiving computer, start the script responsible generating ARP replies
3. On the receiving computer, start the capture file
4. On the transmitting computer, start the script that will perform the injection of the packets into the switch
5. Once all the packets have been injected, stop the capture application on the receiving computer

Once all the experiments were carried out, scripts written in Python were run on the captured files to preform the analysis. Two scripts were written for analyzing the data: one script responsible for determining the inter-packet delay, and the other one for determining the penalty associated with a cache miss. The first script, simple took the difference of capture time between two consecutive packets, and wrote these values out to a comma separated file. The second script performed two calculations, the first being the time difference between two ARP requests, and the second being the time difference between an ARP reply and the arrival of the TCP packet. The workings of the second script are summarized below:

- Run three separate tshark commands, one returning tuples of ARP requests only, one returning ARP replies only, and one returning TCP packets only, storing each one in a different list
- Iterate through the ARP requests list, calculating delta t, and write it to comma separated file. This would represent the first part of the penalty

- Iterate through the ARP replies, and create a dictionary keyed by the destination IP address, and have the transmission time as the value for that key
- Iterate through the TCP packets, and do a lookup in the dictionary lookup the destination address. Subtract the ARP reply's transmission time from the TCP packet's arrival time, and write that value to a comma separated file. This gives us the second part of the penalty. Creating a dictionary made the lookup in the ARP reply list *O(1)*, which is a lot more efficient than having two loops matching the ARP reply with the TCP packet.

The scripts described are included in Appendix A, section A1.

3.2.6 Software Used

A number of applications and libraries were used to complete this project, including Wireshark, Tcpdump, Tshark, Capinfos, and Scapy.

Wireshark, an open source tool available on all common platforms, provides a user interfaced application capable of capturing live Internet traffic on a computer, and allows the viewing of captured file (.pcap files). When viewing previously captured files, the user has the ability to view all the relevant information stored during the capture. Through a user interface, the user can specify a filter for viewing packets that match a specific criteria - instead of viewing all of the captured packets.

Tcpdump is a powerful command-line packet analyzer, which can capture packets and store them in .pcap files. The program is flexible enough to allow the user to set a criteria for the packets to be stored. For example, the user has the ability to specify: the layer of the packets they want stored, which is useful if you have a lot of traffic flowing through an interface, but you are only interested in a subset of that traffic.

TShark is a network protocol analyzer, allowing the user to capture packet data from a live network, or read packets from a previously saved capture file. It also gives the user the flexibility of setting filters for the traffic being captured. Tshark can also be used for capturing files. For the purposes of this thesis work, it was used to to run commands through the Python script to extract information from the capture files.

Capinfos is a program that takes a capture file(s) as input, analyzes its contents, and returns useful statistics about it. Some of the useful information that can be extracted from a capture file includes: the number of packets captured, the capture duration, average data rate, and the average packet rate. For the purposes of this project, it was used to

verify the total number of packets captured, to verify that all the packets have made it, and get the statistics required to calculate the transmission rate.

Finally Scapy is a library that works with Python scripting, giving the user access to built in functions for generating, sending, and receiving packets. It also gives the user the ability to do socket programming - a necessity for achieving high packet transmission rates.

A summary of the software and libraries used is summarized below:

Software/Library	Usage
Wireshark	Viewing .pcap files
Tcpdump	Capturing packets and storing them in .pcap files
Capinfos	Produce statistics on .pcap files
Tshark	Extract specific information from capture files
Scapy	Generate packets, and transmit and receive packets

Table 3.5: A summary of the applications and libraries used to complete the thesis work.

Chapter 4

Results And Analysis

The original intent of this project involved characterizing the data collected from CAIDA, and measuring the performance of the switch as the traffic is shaped towards that of CAIDA's characteristics. It was thought that the switch had a replacement algorithm for the TCAM, and that once the cache reached its capacity, a certain type of traffic would significantly hamper the performance of the switch due to constant swapping of entries in the switch. Unfortunately it was discovered that there is no replacement algorithm for the cache, instead, the switch would just perform Software Switching once the cache reached its capacity and there is a miss in the cache dedicated for Hardware Switching. Initially, the idea was to start with symmetrical traffic that is evenly distributed across a set of destination IP addresses, and then gradually change the characteristics of the traffic to reach that of the traffic data that was analysed. The hope was that this setup would reveal useful information about the scenario under which the switch would have constant misses in the TCAM, and would have to constantly replace entries in the cache.

Eventually, the intent of the project became focused on determining the penalty associated with a cache miss, and the penalty associated with having to do Software Switching once the TCAM became full - shown in experiments 2 and 3 respectively.

4.1 Internet Traffic Characterization

In this section, a sample of the traffic characterization done on the data collected by CAIDA is presented. The data was processed using a Python script included in Appendix AA, section A3. For each characteristic analyzed, the average, standard deviation, variance,

maximum value, and minimum value are calculated. Information analyzed from CAIDA's Internet traffic sample, such as the flow size and average inter-packet delay, would've been used to give our experiments some structure. That is, given that we have control over the flow size, and we can vary the transmission rate, it would be possible to achieve traffic that has similar characteristics of the traffic collected by CAIDA. The data below is grouped by capture files, where two capture files were analyzed.

The first graph below, shows the flow sizes from the first capture file in the data collected by CAIDA:

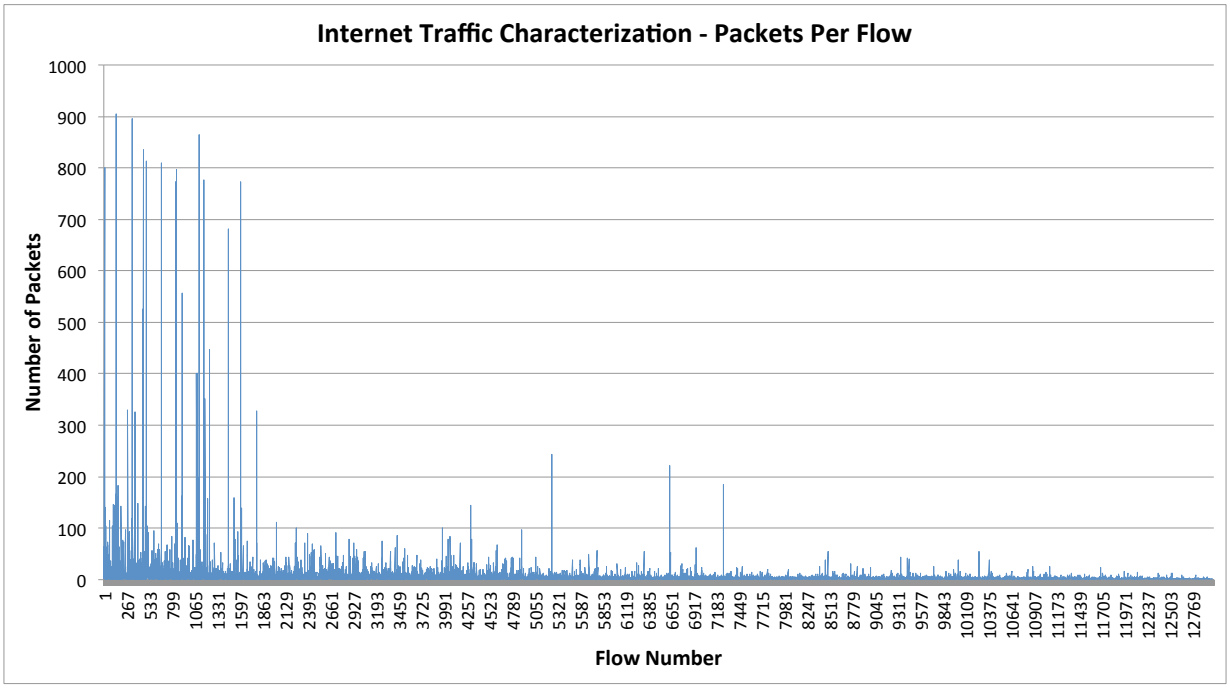

Figure 4.1: This is a sample of the flow size in the first capture file that was analyzed from CAIDA's data set. A flow is defined as a stream of packets with the same source and destination IP addresses, and the same source and destination TCP ports.

In this sample, there are over twelve thousand flows analyzed, in a time period of approximately one minute of capture time. The maximum flow size in this particular data set peaks at 905 packets, while the average is around six packets. The variance is high indicating that the distribution of the values from the mean is large. The standard deviation is also fairly high, signifying that the values are spread out over a large range of values. Below is a table summarizing the statistics on flow size:

Calculation	Packets Per Flow
Max Value	905
Min Value	1
Average	5.60
Standard Deviation	29.19
Variance	852.01

Table 4.1: Summary of the statistics of flow size, for the previously plotted data. This data set is from the first capture file analyzed.

The second graph below presents a sample of average inter-packet delay from the data collected by CAIDA:

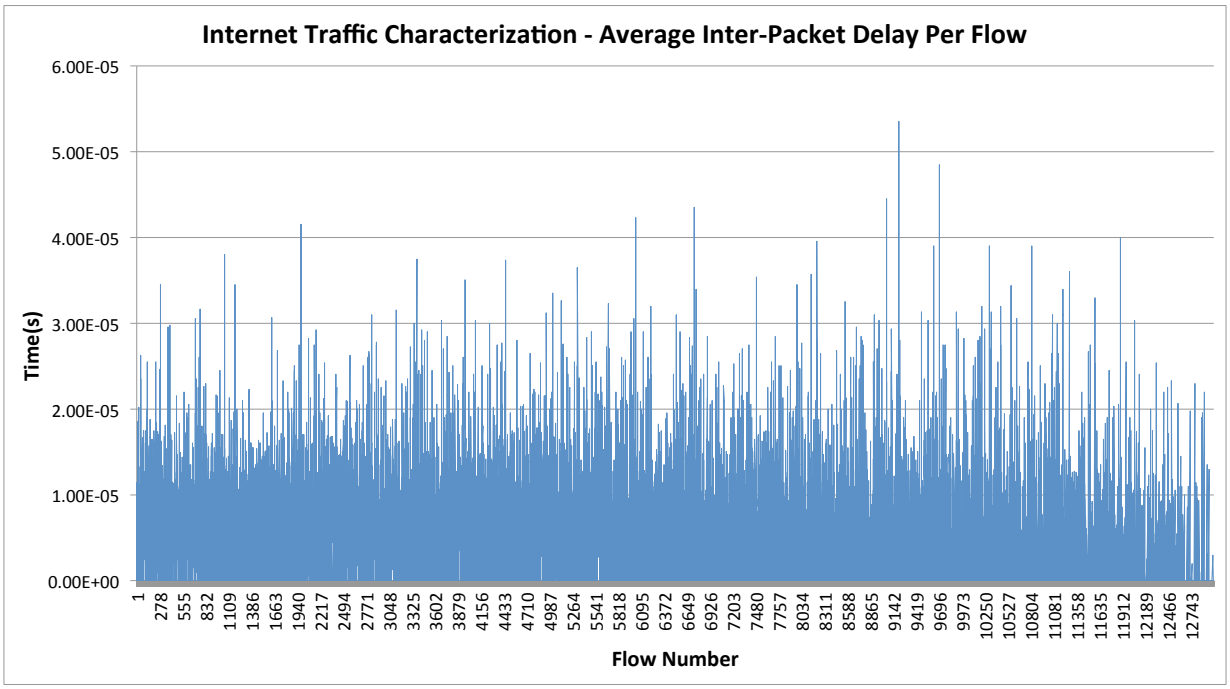

Figure 4.2: This graph represents the per flow average inter-packet delay for the first sample analyzed from CAIDA's data set. The per flow average inter-packet delay is calculated by summing the inter-packet delay for each flow, and dividing by the flow size.

The maximum per flow inter-packet delay in this set, peaks at approximately $53\mu s$, while the average inter-packet delays is approximately $6\mu s$. The standard deviation is low, indicating that the values in the data set are close to the mean. The variance is also low, signifying that the distribution of values from the mean is small, making the mean a good approximation of the data set. A summary of the previously discussed statistics is shown below:

Calculation	Average Inter-Packet Delay (s)
Max Value	5.35×10^{-5}
Min Value	0
Average	5.33×10^{-6}
Standard Deviation	6.31×10^{-6}
Variance	3.98×10^{-11}

Table 4.2: This is a sample of the per flow inter-packet delay in the first capture file that was analyzed. Inter-packet delay for packets in a flow are calculated by taking the difference in arrival time of two consecutive packets. The per flow average inter-packet delay is then calculated by summing the inter-packet delays for each flow, and dividing by the flow size.

The third graph presents a sample of the flow duration from the first capture file collected by CAIDA:

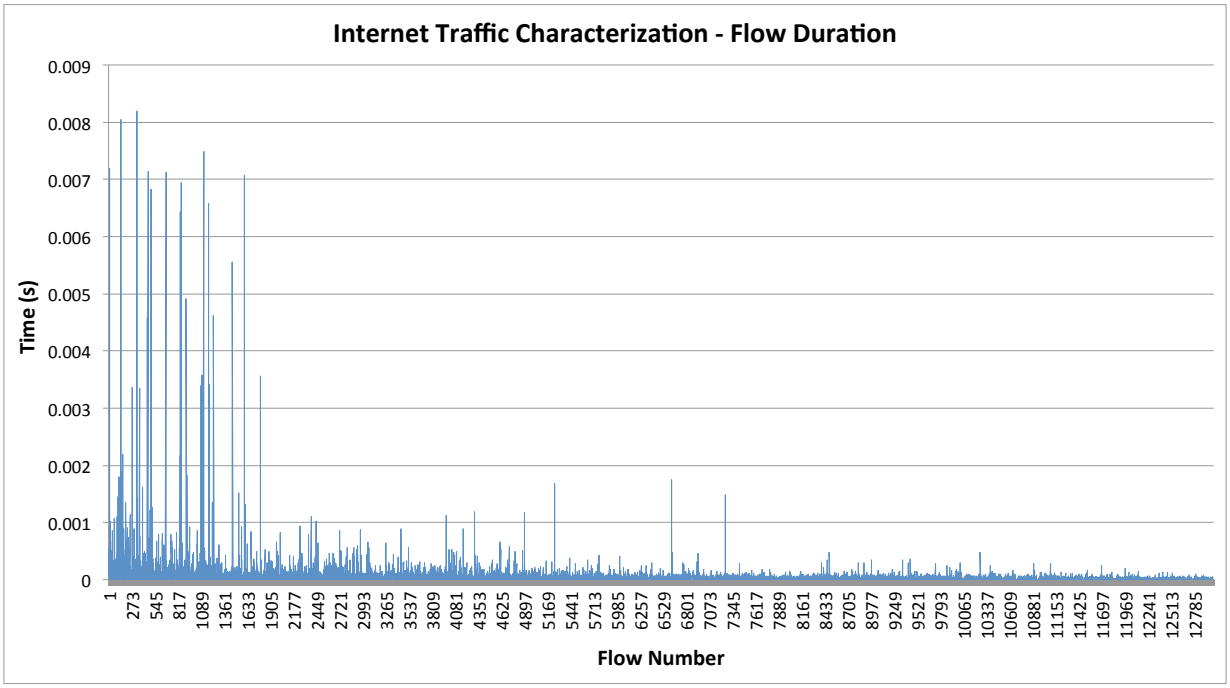

Figure 4.3: This is a sample of the flow duration from the first capture file that was analyzed. The flow duration is defined as the difference between the arrival time of the first packet that arrived in the flow, and the arrival time of the last packet that arrived in the flow.

The maximum flow duration is approximately $8ms$, while average flow duration is approximately $54\mu s$. The standard deviation in this case is high, signifying that the values are spread out over a large range of values, but the variance is small indicating that the values are close to the mean. Below is a summary of the statistics previously discussed:

Calculation	Flow Duration (s)
Max Value	8.19 x 10^{-3}
Min Value	0
Average	5.39 x 10^{-5}
Standard Deviation	2.59 x 10^{-4}
Variance	6.70 x 10^{-8}

Table 4.3: Summary of the statistics of the flow duration for the data previously plotted from the first capture file analyzed.

For the purposes of comparing different sets of data, the analysis of another capture file is also included. Below is a sample of flow size from the second capture file:

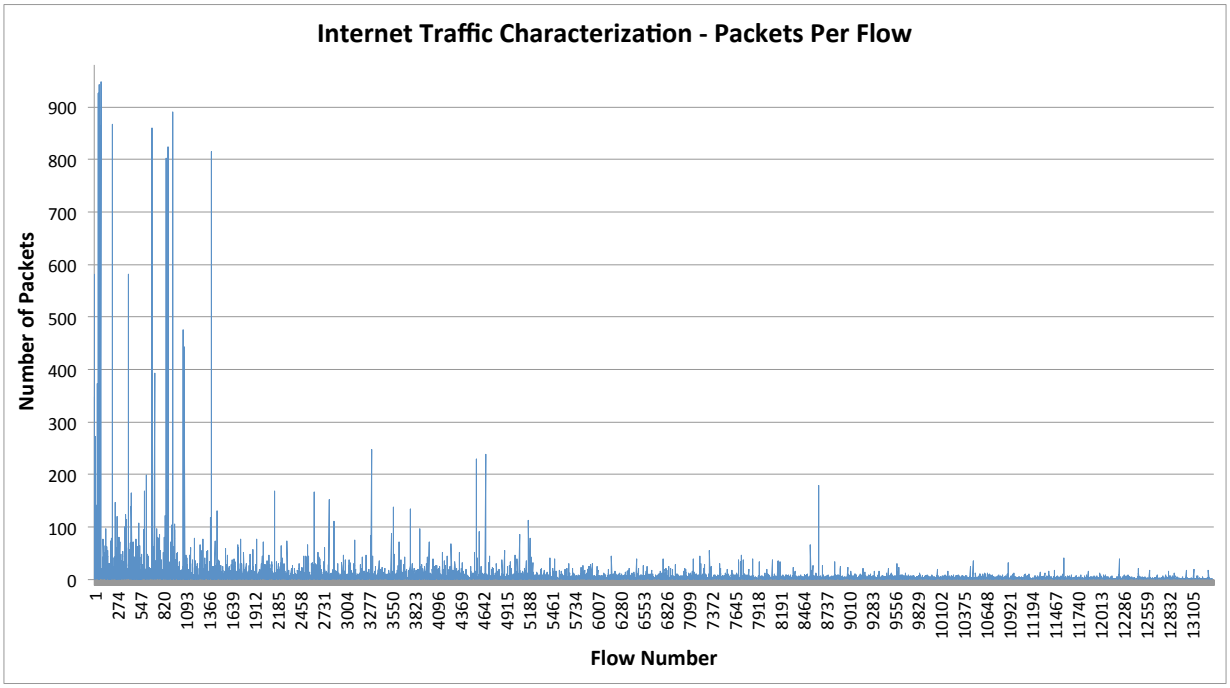

Figure 4.4: This is another sample of the flow size, form a different CAIDA data set. A flow is defined as a stream of packets with the same source and destination IP addresses, and the same source and destination TCP ports.

In this sample there are over thirteen thousand flows analyzed. The statistics of this set are very similar to the previous set of data, where the maximum value is peaking at 947 packets per flow, and an average of 5.70 - slightly higher than the previous set. The variance and standard deviation are also fairly high, indicating that the distribution of the values from the mean is large, and that the values are spread out over a large range of values, respectively. A table summarizing the statistics previously discussed is shown below:

Calculation	Number of Packets Per Flow
Max Value	947
Min Value	1
Average	5.70
Standard Deviation	28.33
Variance	802.80

Table 4.4: Summary of the statistics of flow size, for the previously plotted data. This data set is from the second capture file analyzed.

A sample of average inter-packet delay from the second capture file is shown below:

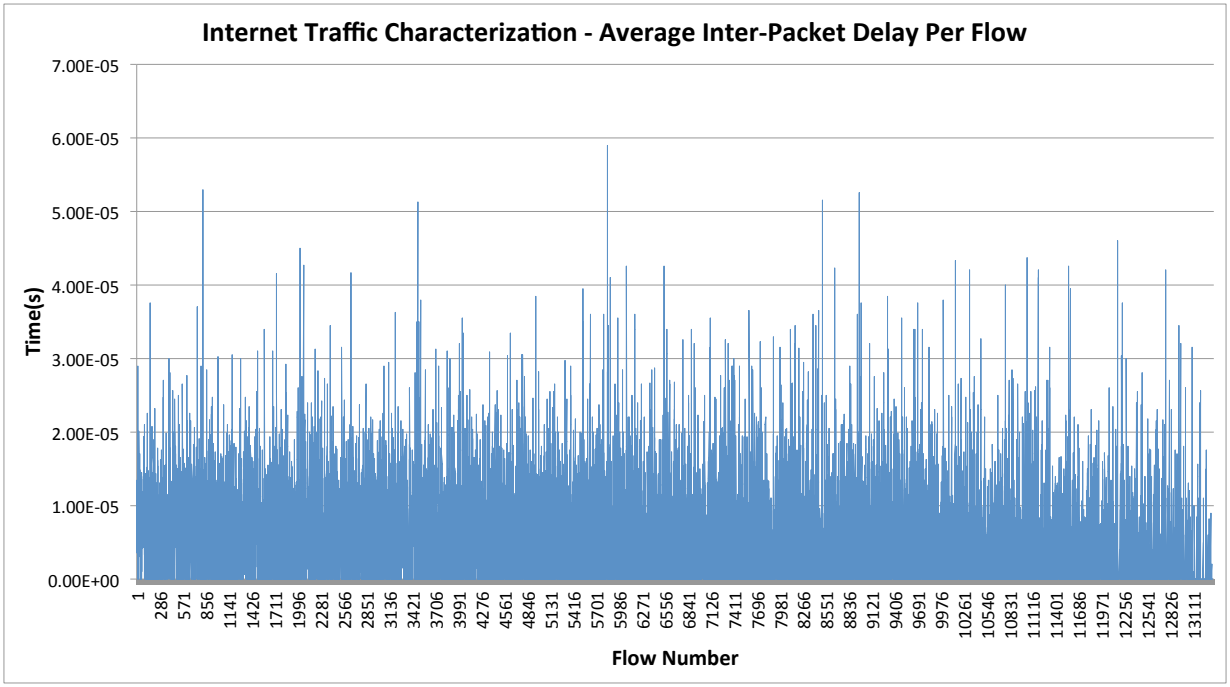

Figure 4.5: This is a sample of the inter-packet delay in one of the capture files that were analyzed. Inter-packet delay is calculated by taking the difference in arrival time of two consecutive packets.

In this case the average inter-packet delay peaks at $59\mu s$, while averaging at approximately $5.7\mu s$. Again in this data set, the variance and standard deviation are both low, indicating that the values are close to the mean and the distribution of the values from the mean is small, respectively. A summary of the statistics discussed, is shown below:

Calculation	Number of Packets Per Flow
Max Value	5.90×10^{-5}
Min Value	0.00
Average	5.71×10^{-6}
Standard Deviation	6.76×10^{-6}
Variance	4.57×10^{-11}

Table 4.5: This is a sample of the inter-packet delay in the second capture files that was analyzed. Inter-packet delay is calculated by taking the difference in arrival time of two consecutive packets.

A sample of the flow duration from the second capture file is shown below:

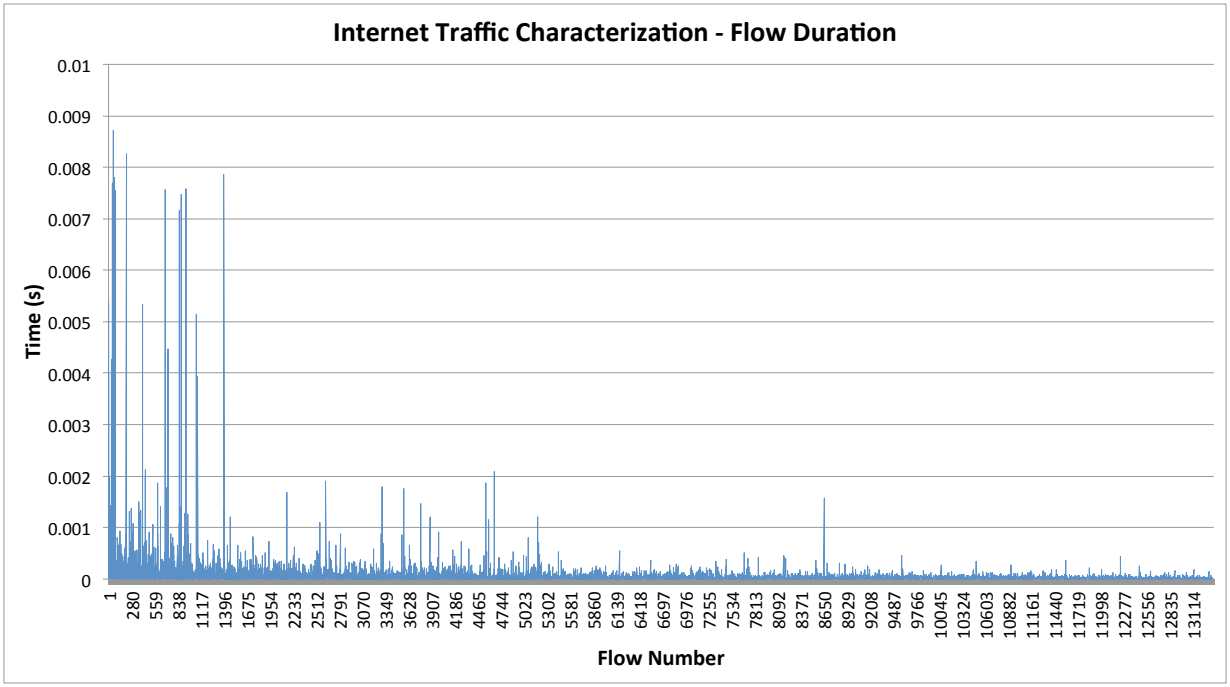

Figure 4.6: This is a sample of the flow duration from the first capture file that was analyzed. The flow duration is defined as the difference between the arrival time of the first packet that arrived in the flow, and the arrival time of the last packet that arrived in the flow.

In this case the flow duration peaks at approximately $8.7ms$, and averaging at approximately $54.5\mu s$. Similarly to the previously analyzed set, the variance and standard deviation are both low, indicating that the values are close to the mean and the distribution of the values from the mean is small, respectively. Below is a summary of the statistics previously discussed:

Calculation	Flow Duration (s)
Max Value	8.73×10^{-3}
Min Value	0
Average	5.45×10^{-5}
Standard Deviation	2.6×10^{-4}
Variance	6.82×10^{-8}

Table 4.6: Summary of the statistics of the flow duration for the data previously plotted from the second capture file analyzed.

Even though the two samples are taken at different times, we can see that the analyzed statistics are very similar. Other samples were analyzed in a similar fashion, revealed that the general trend of the statistics. Other characteristics that could've been analyzed are: the distribution of the destination IP addresses and the flow rate, but considering the large data set that was available, determining the distribution of destination IP addresses would've been very difficult to do.

4.2 Experimental Results

A few experiments were run for this project to benchmark the performance the switch. The purpose of these experiments was to determine the penalty associated with a miss in the TCAM cache and the penalty associated with performing Software Switching, versus doing Hardware Switching. The first two experiments involved determining the penalty incurred when a miss occurs in the cache while performing Hardware Switching and the last experiment is used to determine the penalty incurred from having to do Software Switching.

4.2.1 Experiment 1

The purpose of the first experiment is to collect information about the inter-packet delay of packets when there is a hit in the cache. This occurs when the packet arriving to the switch has a destination address that is stored in the cache. This experiment was performed first because the values obtained from this experiment is need for some of the calculations done in the second experiment to determine cache miss penalty.

A high level diagram depicting the operation of the switch while its performing Hardware Switching for two packets with destination IP addresses that have an entry in the cache is shown below:

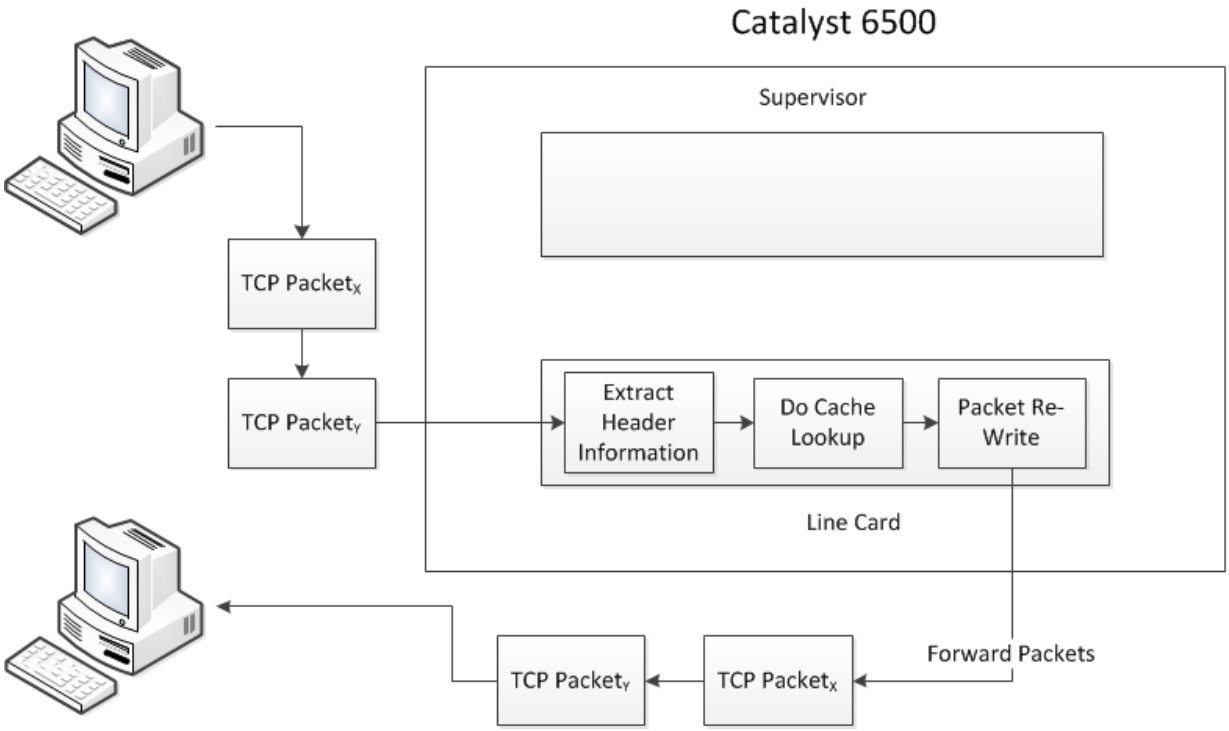

Figure 4.7: This is the scenario when the switch is performing Hardware Switching, and the TCP packet that is being processed has a destination IP address that is in the cache.

Note that in this case, the supervisor is not involved at all - unlike the case where there is a cache miss. Ideally, for the majority of the incoming TCP packets, you want them to have a hit in the cache, to obtain maximum throughput. Any time the supervisor has to get involved to make a forwarding decision due to a miss, or to perform Software Switching, the throughput of the switch decreases.

Below is a sample graph of the inter-packet delay of packets with a hit in the cache:

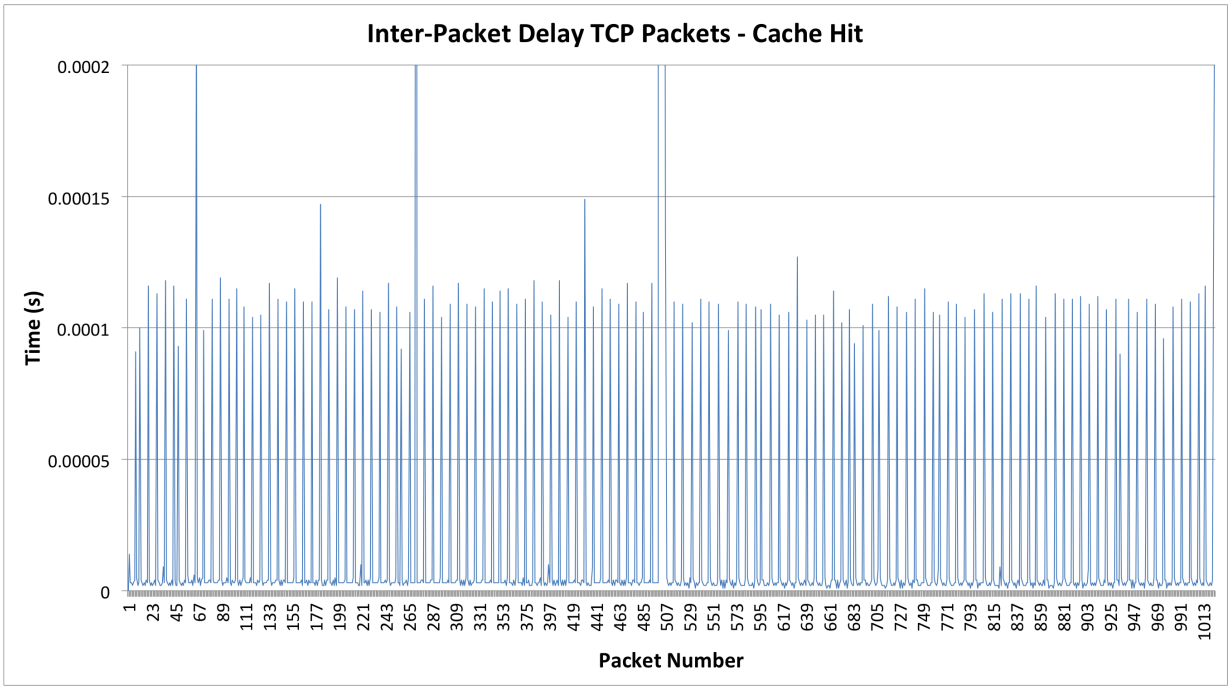

Figure 4.8: This is the inter-packet delay of TCP packets arriving at the receiver. All the packets in this set have destination IP addresses that are in the switch's cache. The inter-packet delay is calculated by taking the difference in time between two consecutive packets arriving at the destination.

The data in the previous graph was obtained by taking the time difference between two consecutive TCP packets from the capture file on the receiver computer. The graph shows a few fluctuations in the time delay, all of which are within the range of $1\mu s$ to $100\mu s$. More information highlighting some information about the plotted data set is shown in the table 4.8 below:

Calculation	Time Delay (s)
Max Value	8.45 x 10^{-2}
Min Value	0
Average	101.99 x 10^{-6}
Standard Deviation	2.64 x 10^{-3}
Variance	6.98 x 10^{-6}

Table 4.7: Inter-packet delay statistics for TCP packets with destination IP addresses that have entries in the cache.

The minimum and maximum values show the range of inter-packet delay of TCP packets that have a destination IP address in the cache - a fairly large range. The standard deviation is relatively high, confirming that the values are spread out over a large range of value. On average, the inter-packet delay of forwarding TCP packets is approximately $102\mu s$. The low variance value indicates that the distribution of the values from the mean is small. Therefore the average value could be used as a good approximation for the amount of time it takes the switch to extract the header information, do a lookup in the cache, and finally do a packet re-write. This value is useful as it will be used in the the calculations of Experiment 2 to remove extra time included in the measurements, but is not part of the penalty of having a cache miss.

It is unclear exactly why there are large fluctuations in the previously plotted data set, but we hypothesis from the consistency in the frequency in which the spikes appear in, that it could be due to some recurring process in the switch. The switch might be doing some house keeping necessary for the operation of the switch, causing interrupts in the processing of packets. A graph, with the outliers removed, is shown below:

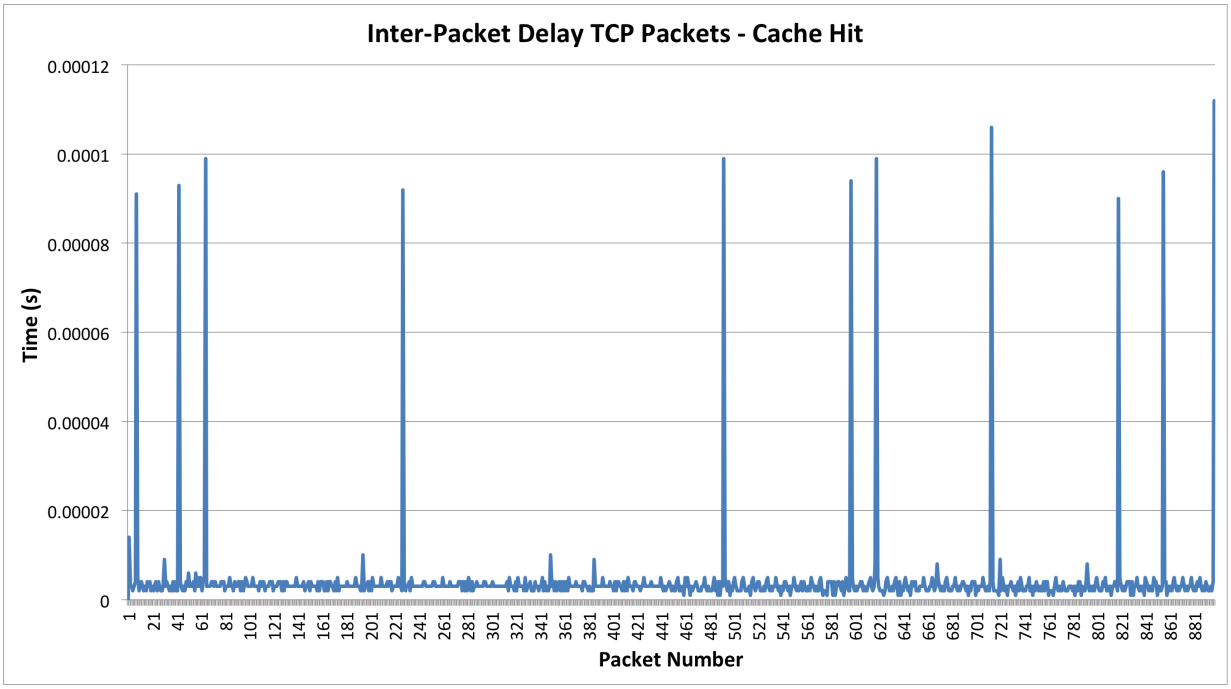

Figure 4.9: This is the inter-packet delay of TCP packets arriving at the receiver with the outlier values causing the spikes in the graph removed. All the packets in this set have destination IP addresses that are in the switch's cache.

In this case, the spike occurs every eighth packet, while in the data sets presented later on, the frequency is different. For this data set, the information regarding the spikes is displayed below:

Statistic	Value
Frequency	Every 8 Packets
Percentage of data causing Spikes	12.5 %
Average	4.23 x 10^{-6}s
Range of values without spikes	1 x $10^{-6}s$ to 112 x $10^{-6}s$
Standard Deviation	10.47 x $10^{-6}s$
Variance	1.1 x $10^{-10}s$

Table 4.8: Statistics about the spikes in the data analyzed for the inter-packet delay of TCP packets during a cache hit.

For the calculations in Experiment 2, we will be using the values from the data set which does not contain the spikes, since it is more representative than the original set.

4.2.2 Experiment 2

The second experiment is aimed at determining the penalty associated with a packet destined to an IP address that is not stored in the cache. The first experiment involved firing packets at the maximum speed possible, with an empty TCAM cache. A miss in the cache involves a penalty that can be split into parts: the time it takes to realize that the information is not in the cache and the time it takes to initiate Address Resolution Protocol, and the second, the time it takes to process an ARP response, add an entry into the cache, and finally forward the TCP packet to the destination. The experiment setup is designed in such a way so as to eliminate any slowdowns that might be caused by the computer. In this case, the final destination of the packet is one hop away from the switch, which is typically the shortest it will ever be. If the destination was more than one hop away, then the propagation delay of the requests, replies, and routing of TCP packets would be increased - thus affecting the data collected.

The first part of the penalty occurs when the switch receives a packet with a destination IP address, which is not in the cache. This causes the switch to forward the packet information to the Supervisor, and the Supervisor sends out an ARP request on the interface that has an IP address and mask that contains the destination IP address in its subnet. For example, if the switch has an interface, *GigabitEthernet8/25*, with IP address 100.8.0.1

and subnet mask 255.255.0.0, all the packets in the range 100.8.0.1 to 100.8.255.255 are taken care of by this interface. So if a packet destined for 100.8.0.100 does not have a hit in the cache, an ARP request is sent out on interface *GigabitEthernet8/25*. If an interface on the switch is configured to be a default gateway, then an ARP is also sent out on that interface.

A high-level diagram depicting the operation of the switch when there is a miss in the cache is shown below:

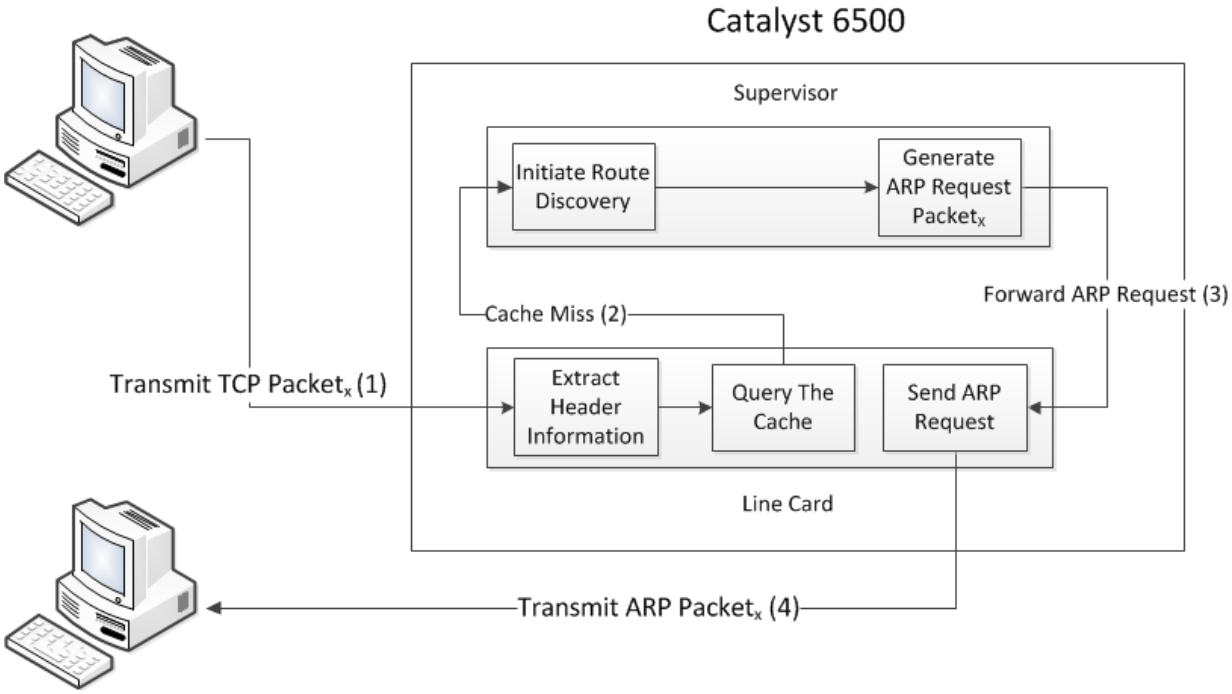

Figure 4.10: This is the scenario where a TCP packet has a destination IP address that is not in the cache, resulting in a miss when a look up is performed while trying to perform Hardware Switching. Note the involvement of the Supervisor, which initiates the Address Resolution Protocol. This diagram constitutes the first part of the penalty associated with a cache miss.

Since the switch sends out ARP requests for all the packets that have destination IP addresses that are not in the TCAM, firing packets consecutively that are not in the TCAM, will allow us to measure the time associated with extracting the header information from the packet, querying the cache, and forwarding the packet to the supervisor which sends out an ARP request. It should be noted that this portion of the penalty also incorporates the time it takes to extract header information from the packet and the time it takes to perform the TCAM lookup - two steps that occur regardless if there is a miss or a hit in the cache.

The following table summarizes the abbreviations involved in the calculation of the first part of the penalty:

Abbreviation	Description
t_{ARPX}	The time when an ARP request packet is sent
t_{ARPR}	The time when an ARP reply packet is sent
t_{ARPP}	The time it takes to process ARP reply
t_{PKTR}	The time it takes to do a packet re-write
t_{TCPA}	The time a TCP packet arrives to the computer
t_{AENT}	The time it takes to add an entry into the cache
t_{MINPD}	The minimum inter-packet delay for packets with a hit in the cache
t_{P_1}	The first part associated with the penalty of a miss in the cache
t_{P_2}	The second part associated with the penalty of a miss in the cache
t_{P_t}	The total penalty for a miss in the cache

Table 4.9: Abbreviations used for in calculations

The equation for calculating the first part of the penalty, where the ARP request packets are numbered based on their order of arrival and $t_{ARPX_2} > t_{ARPX_1}$, is shown below:

$$t_{P_1} = t_{ARPX_2} - t_{ARPX_1} \tag{4.1}$$

A sample of the data collected for the first part of the penalty, which is the time it takes to extract the packet information header information, query the TCAM, discover a miss, construct an ARP request and send it out is shown below:

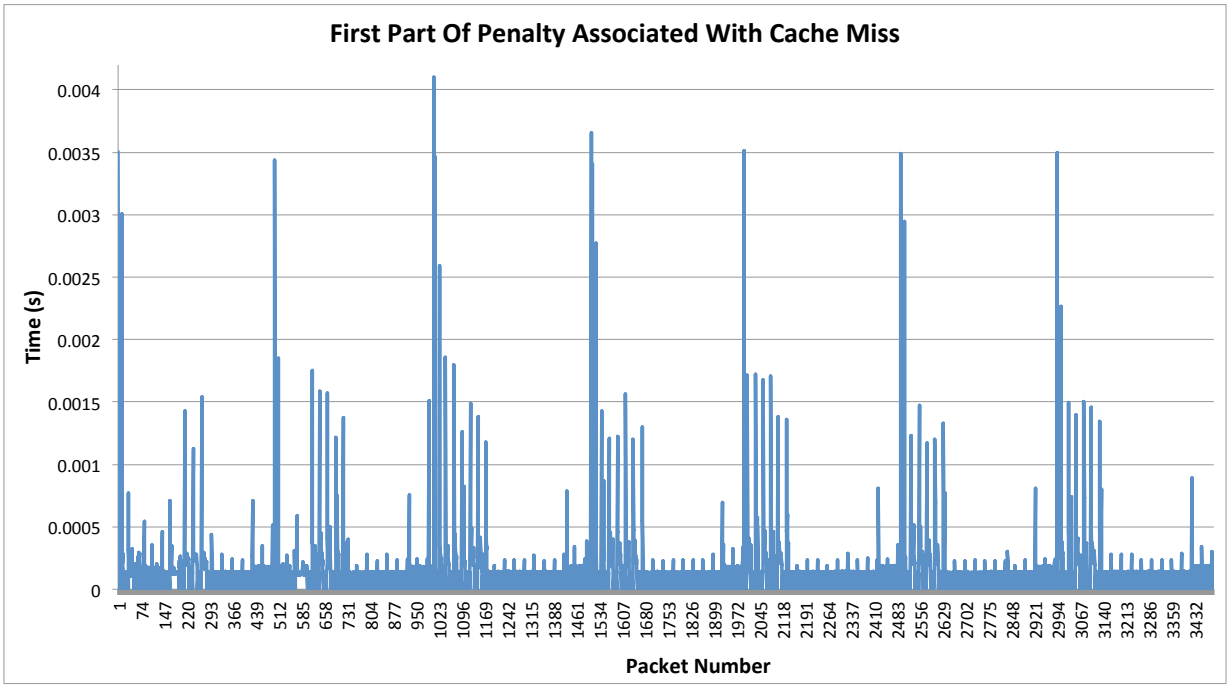

Figure 4.11: The penalty associated with extracting the header information, performing a cache lookup, generating an ARP request, and transmitting the ARP request.

Figure 4.11 has a pattern which repeats every approximately 500 packets. For the first batch of packets, there are a few fluctuations in the time delay, which could be attributed to the switch working towards a transient state, or the computer ramping up its transmission rate. From the graph shown, the majority of the the packet delays are in the range of *2ms*. For better viewing, a version with a smaller subset of the figure 4.11 is shown below:

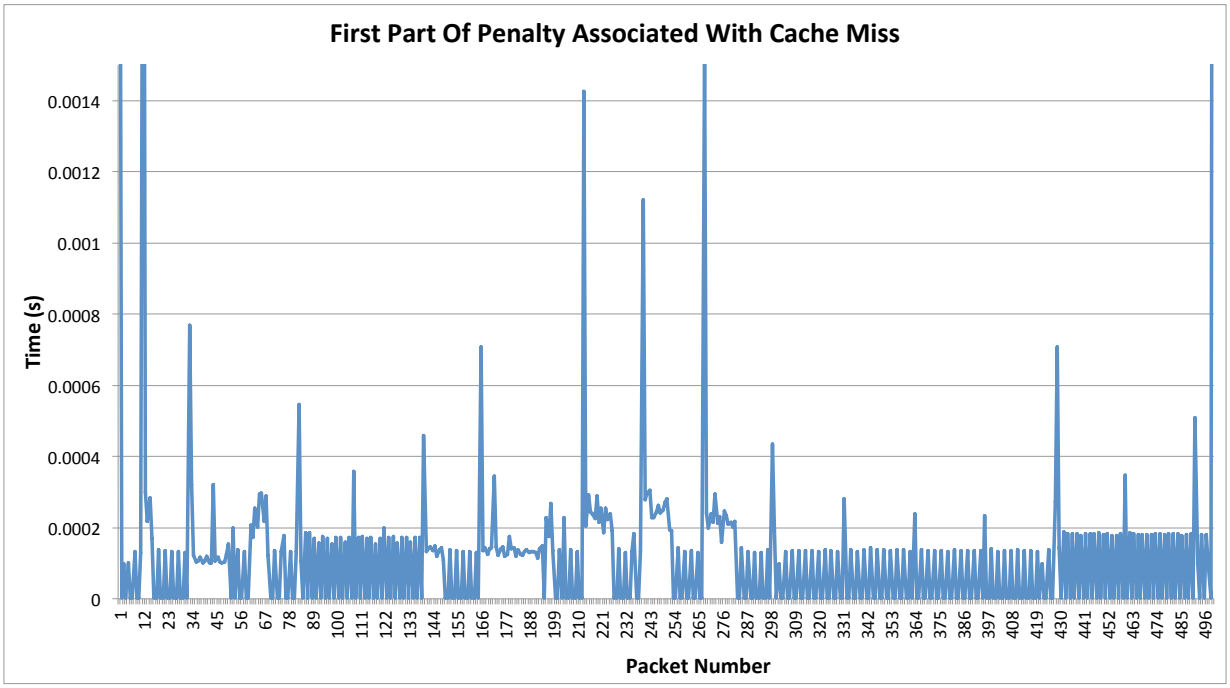

Figure 4.12: This graph contains a smaller subset of graph 4.11, showing data up to the end of the first sequence that repeats. The range of the y-axis has also been adjusted to have a better focus on the majority of the data points.

Analysis of the data shown in the graph is included in table 4.10. The table includes basic calculations of the average, standard deviation, variance, and the maximum and minimum values of the set used to plot the graphs. The analysis shows that on average, the first part of the penalty is approximately $117\mu s$ - a significant penalty that should not be incurred frequently. Of course, when the switch is first installed, or if the switch crashes and restarts, this penalty is incurred a few times until all the surrounding addresses are learned.

Calculation	Time Delay (s)
Max Value	4.11×10^{-3}
Min Value	2×10^{-6}
Average	117.39×10^{-6}
Standard Deviation	$271.17\ 10^{-6}$
Variance	7.38×10^{-8}

Table 4.10: Statistics for the penalty associated with extracting the header information, performing a cache lookup, and generating an ARP request.

The table above 4.10 shows that the data set has a low variance and standard deviation, even though there are a few spikes in the data. Again in this case, there are periodic spikes, that we hypothesis to be due to some processes running in the switch that interrupt the switching process. In this case, the spikes follow a rigid pattern as they did in the data collected in the first experiment, where it seems to repeat every 500 packets. Some statistics about the spikes are shown below:

Statistic	Value
Percentage of data causing spikes	1.63 %
Average	8.88 x 10^{-5}s
Range of values without spikes	2 x $10^{-6}s$ to 797 x $10^{-6}s$
Standard Deviation	298.95 x $10^{-6}s$
Variance	1.23 x $10^{-8}s$

Table 4.11: Statistics of the data set after removing the spikes in the penalty associated with performing a cache miss

Note that after removing the spikes in the graph, the data is more representative. The variance and standard deviation have been significantly reduced, making the new average a more accurate approximation. The graph below contains a zoomed in version of the data with the majority of the spikes removed, plotting values that are within the time delay range discussed in the previous table - where over 98 percent of the data lies:

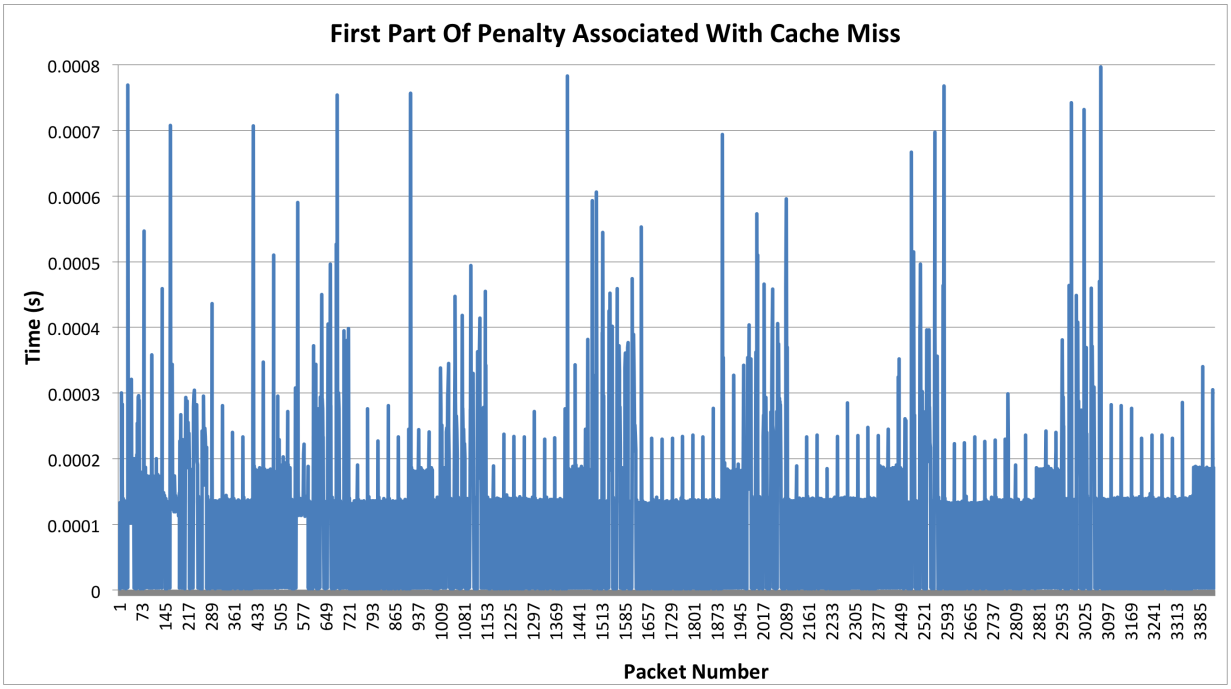

Figure 4.13: Graph displaying the penalty associated with extracting the header information, do a cache look, and generating an ARP. This dataset has the spikes in the time delay exceeding the $800\mu s$ removed.

Upon closer examination of the previous calculation for the first part of the penalty, it was concluded that this calculation can be refined further by removing the time it takes to perform some of the common steps between processing a packet that has a destination IP address with a hit in the cache, and one which will have a miss in the cache. There are four steps that the switch performs when processing a packet, if there is a miss in the TCAM 4.10

1. Extract header information from the packet
2. Query the TCAM for destination address
3. Send the header information to the supervisor for a decision
4. Supervisor generates an ARP request and passes it to the line card
5. Line card sends the ARP request out on the appropriate interface

The first two steps mentioned above will occur when the switch is doing Hardware Switching, regardless of whether there will be a hit or a miss in the cache. We can also assume that the amount of time it takes to do a packet re-write for TCP packets that have hit in the cache, is equal to the amount of time it takes to pass the packet from the Supervisor to the line card for transmission. Therefore, assuming that the transmission rate is the same in both cases, subtracting the inter-packet delay of packets that have destination IPs in the cache, should yield a good approximation of the amount of time it takes to process a cache miss, plus the time it takes to generate an ARP request. The following abbreviations are used to calculate a more accurate value of the first part of the penalty:

Abbreviation	Description
t_{EX}	The time to extract the header information
t_{L}	The time to do a lookup in the cache
t_{FWD}	The time to make a forwarding decision
t_{ARP}	The time it takes to create an ARP request
t_{PRTM}	The total time it takes to process a packet with a miss
t_{PRTH}	The total time it takes to process a packet with a hit

Table 4.12: Abbreviations for calculating a more accurate calculation of the first part of the penalty.

If the arriving TCP packet has a destination IP address in the cache, then the total processing time would be:

$$t_{PRTH} = t_{EX} + t_L = t_{MINPD} \tag{4.2}$$

If the arriving TCP packet has a destination IP address that is not in the cache, then the total processing time would be:

$$t_{PRTM} = t_{EX} + t_L + t_{FWD} + t_{ARP} \tag{4.3}$$

The equation for the refined calculation of the first part of the penalty, where the ARP request packets are numbered based on their order of arrival and $t_{ARPX_2} > t_{ARPX_1}$, is shown below:

$$t_{P1} = t_{ARPX_2} - t_{ARPX_1} - t_{MINPD} \tag{4.4}$$

From the previous equations, subtracting the minimum inter-packet delay of TCP packets that have information about the destination in the cache, will remove the time it takes to do the header extraction (t_{EX}) and the time it takes to do the cache lookup (t_L).

A sample of the refined penalty time for a miss in the cache is shown below with the original calculation being the line in blue, and the refined calculation being the line in red. A version containing a smaller subset for better viewing purposes is also shown.

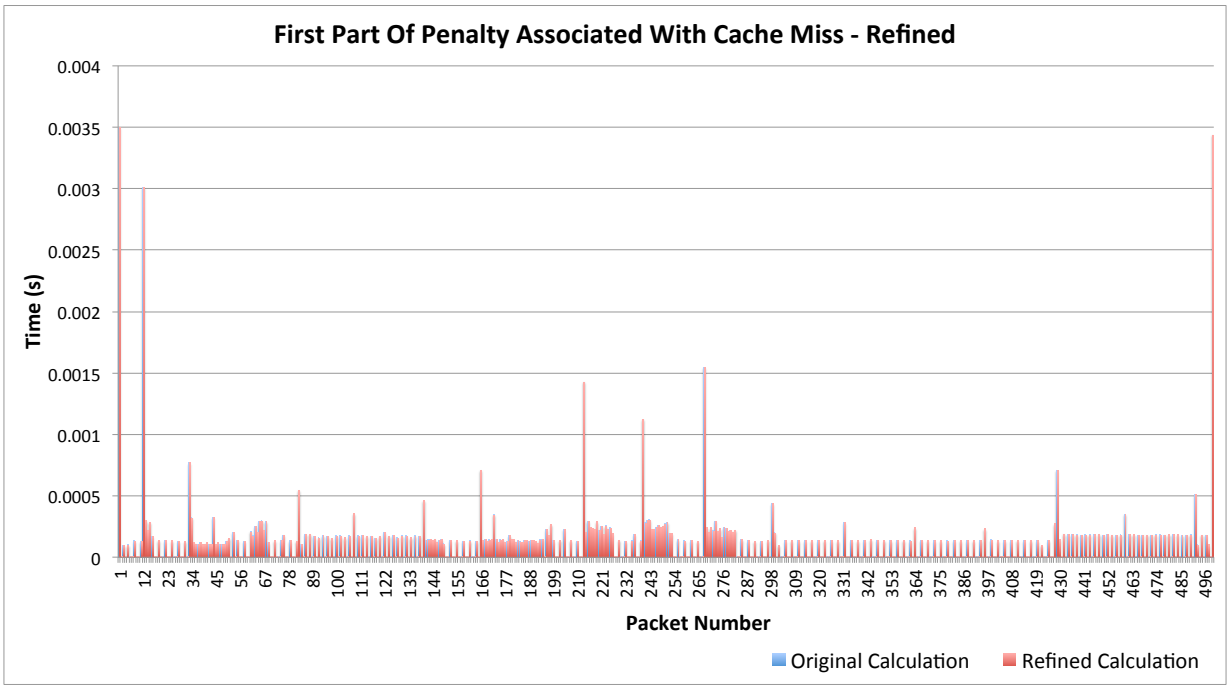

Figure 4.14: Graphing the original penalty versus the refined calculation associated with extracting header information, performing a cache lookup, generating an ARP request, and transmitting the ARP request. The refined calculation aims to remove the time it takes to extract the header information and perform a cache lookup.

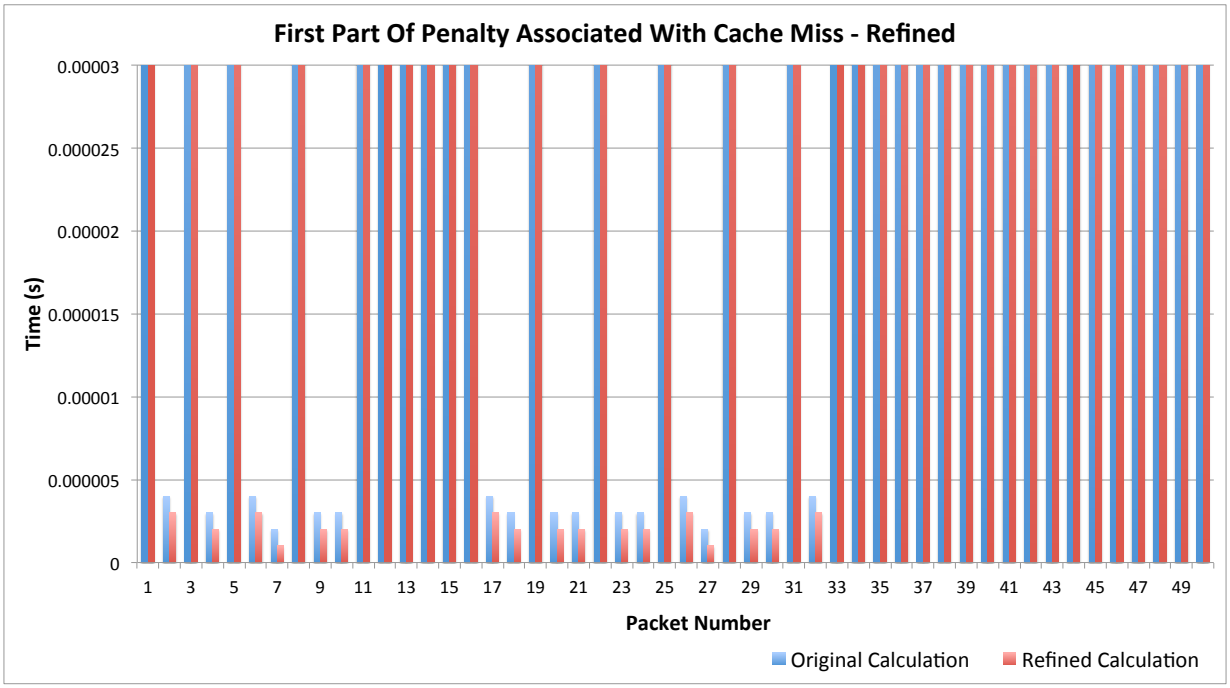

Figure 4.15: A zoomed-in version of the previously shown graph, showing the original penalty versus the refined calculation associated with extracting header information, performing a cache lookup, and generating an ARP request. The refined calculation aims at removing the time it takes to extract header information, and perform a cache lookup. This is a smaller set to magnify the difference in values.

The first figure 4.14 has a pattern which repeats approximately every 500 packets. For better viewing, a magnified version of the first figure is shown below it 4.15 to show the variation in time between the original penalty calculation, and the refined version. The data was plotted in bar graph format because in many cases, the difference between the original calculation and the refined one, is very small. From the magnified version of the graph, it is clear that the difference between the two calculations is very small, and thus had to be plotted as a bar graph. Note that the data between the two calculations varies by $1\mu s$ - the minimum amount of time it takes to extract the header information and do a cache lookup. This result is consistent with what is expected of the switch - the cache lookup and packet re-write has to be extremely fast. Further analysis of the data is shown below:

Calculation	Value (s)
Max Value	4.10 x 10^{-3}
Min Value	1 x 10^{-6}
Average	116.63 x 10^{-6}
Standard Deviation	273.68 x 10^{-6}
Variance	7.49 x 10^{-8}

Table 4.13: First Part Of The Penalty Statistics

The statistics in table 4.13 show a small change in the data, after subtracting t_{MINPD}. The average decreased by $1\mu s$, while the standard deviation and variance remained the same. The maximum and minimum values also decreased by 1μs. Since the time difference is so small between the original penalty calculation and the refined one, this confirms that the switch performs the extraction of the header information and perform a cache lookup extremely fast.

The second part of the penalty occurs when the switch receives the ARP reply from a destination IP address that was previously unknown. The penalty involves the processing of the ARP reply, which involves adding an entry into the cache and sending the TCP packet that might still be in the buffer.

A high-level diagram depicting the operation of the switch when it is processing the ARP reply is shown below:

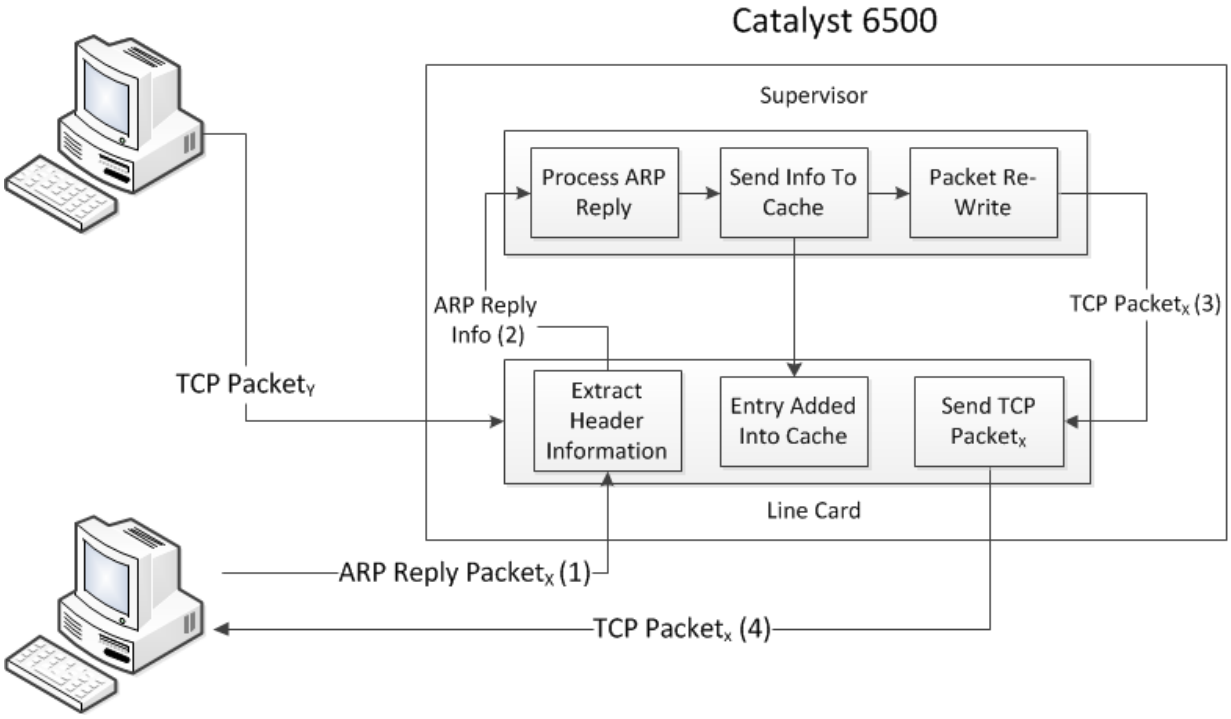

Figure 4.16: Operation of the switch when processing an ARP reply. This is the second part of the penalty, which calculates the time difference between the ARP reply and the arrival of the TCP packet.

To measure this portion of the delay, we have to add up the time it takes to process the ARP reply, add an entry into the cache, and do a packet re-write to forward the packet. This can be represented using the following equation:

$$t_{P_2} = t_{ARPP} + t_{AENT} + t_{PKTR} \tag{4.5}$$

To measure this portion of the delay using the information we have, we take the difference in time between the arrival of the TCP packet, and the time the ARP reply was transmitted for that TCP packet. Upon closer examination of figure 4.16, we can see that the difference in time between when the packet leaves the receiving computer, and the arrival of the TCP packet, is the time it takes to process the ARP reply and forward the TCP packet. The equation for calculating the second part of the penalty is shown below:

$$t_{P_2} = t_{TCPA} - t_{ARPR} \tag{4.6}$$

Below is a graph 4.17 showing the results of the second part of the penalty, with a magnified version 4.18 below that:

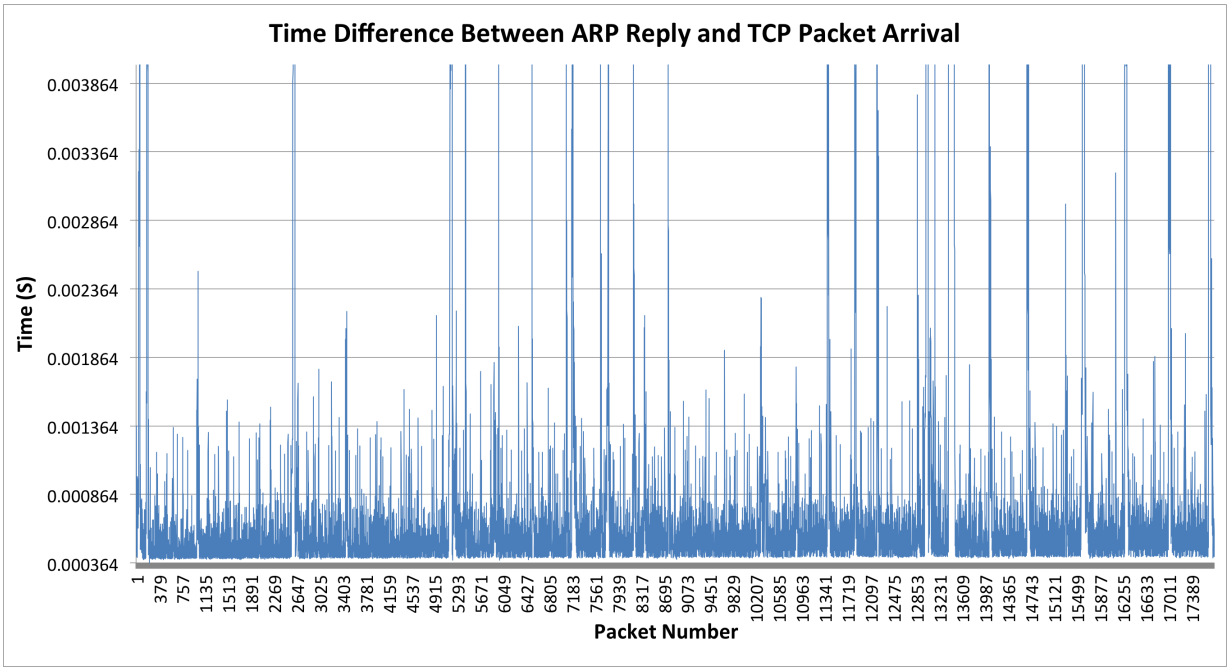

Figure 4.17: This graph represents the second part associated with the penalty of a miss in the cache. This data is calculated by taking the difference of time between the ARP reply sent out by the receiver computer, and the arrival time of the TCP packet.

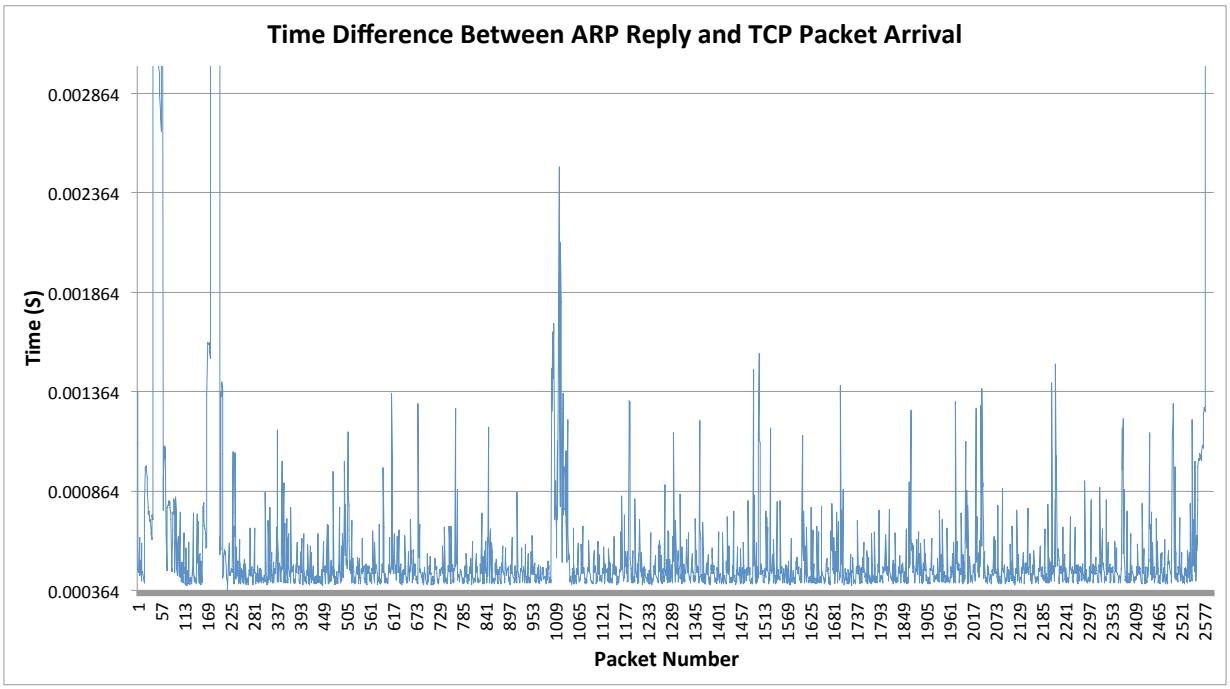

Figure 4.18: A zoomed-in version of the previous graph, showing the second part associated with a penalty.

The second part of the penalty is significantly higher than the first part of the penalty - on average seven times larger. This result makes sense because the second part of the penalty involves more steps, and thus, it is expected that it would take a longer time. The first part of the penalty is in the range of $100\mu s$, while the second part is in the range of $700\mu s$ - a significant difference.

Further analysis of the data is shown below:

Calculation	Time Delay (s)
Max Value	64.6 x 10^{-3}
Min Value	364 x 10^{-6}
Average	757.15 x 10^{-6}
Standard Deviation	1.55 x 10^{-3}
Variance	2.41 x 10^{-6}

Table 4.14: Statistics of the second part of the penalty - the amount of time it takes to process an ARP reply, and forward the TCP packet.

Again the range of values for the delay are significant - starting at $364\mu s$ and ending at $64ms$. On average, the penalty incurred from processing an ARP packet, adding an entry into the cache, and forwarding the TCP packet, is approximately $757\mu s$. Since the variance is fairly low, the average could be considered to be a good representation of the bulk of the data collected.

Summing the two parts of the penalty, gives us the total penalty associated with a cache miss. This setup and methodology, minimizes any delays that are introduced by the computers involved in transmitting and receiving the packets. While a few iterations have been made to the software in order to improve its run time, the computer responding to ARP replies still had a certain threshold that could not handle extremely high packet transmission rates. The equation to determine the penalty associated with a cache miss for routing packet x is shown below:

$$t_{PT} = t_{P1} + t_{P2} \tag{4.7}$$

Finally, the graphs 4.19 and 4.20 below show a graphical representation of the total penalty:

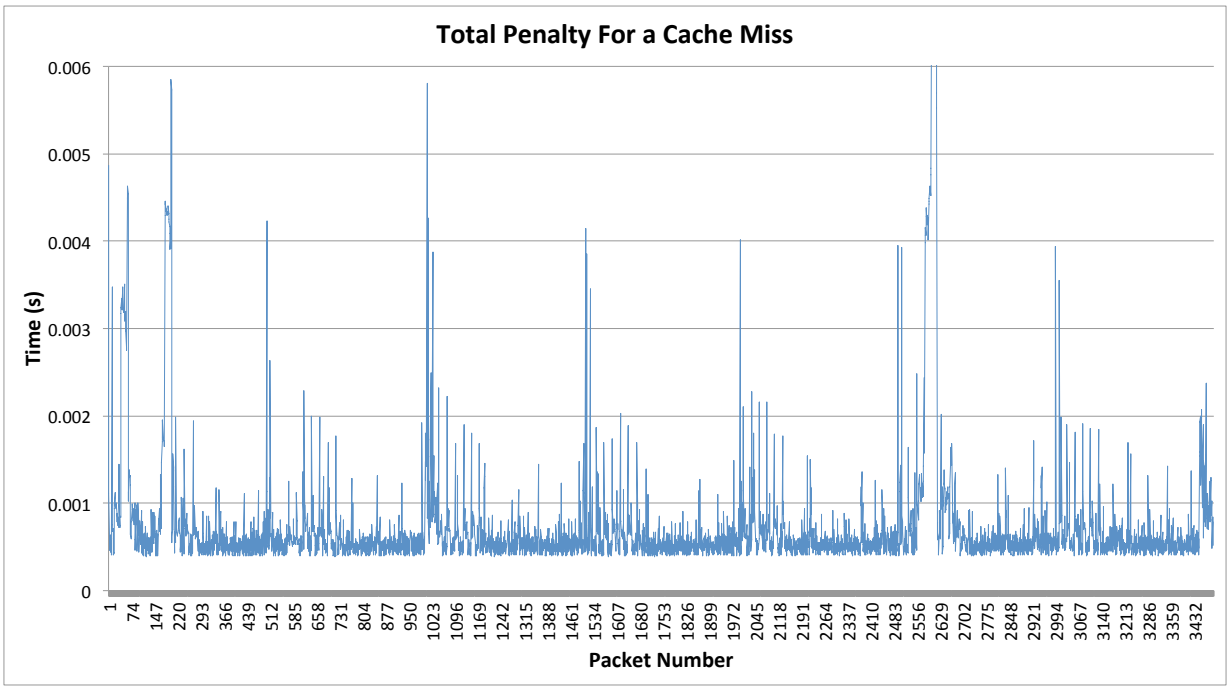

Figure 4.19: Total penalty for a cache miss - a summation of the amount of time it takes to generate an arp request, process an arp reply, and adding an entry into the cache.

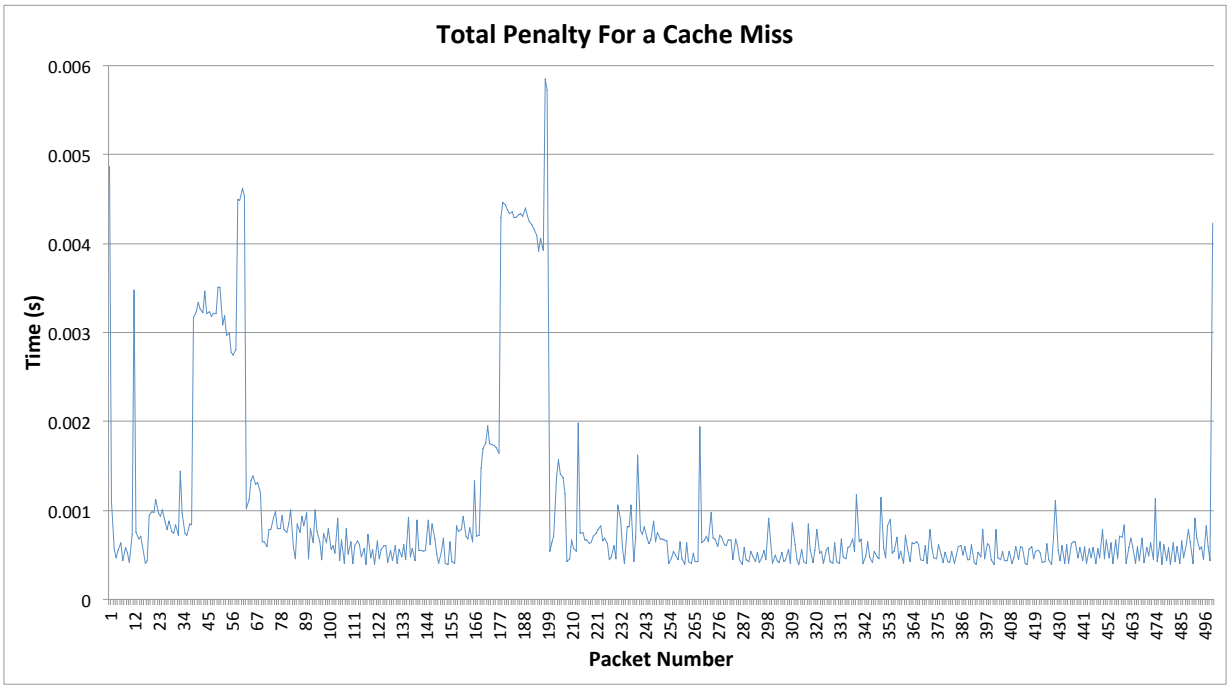

Figure 4.20: A zoomed in view of the graph 4.19, showing a smaller sample of the total penalty associated with a cache miss.

Further analysis of the data is shown below:

Calculation	Time Delay (s)
Max Value	8.73 x 10^{-3}
Min Value	387 x 10^{-6}
Average	738.17 x 10^{-6}
Standard Deviation	760 x 10^{-6}
Variance	5.8 x 10^{-7}

Table 4.15: Statistics of the total penalty associated with a miss in the cache.

Therefore, on average, the penalty of a cache miss is approximately 738μs. The high standard deviation and variance stem from having the same characteristics in the data collected from the first part of the penalty.

4.2.3 Experiment 3

The purpose of this experiment is to demonstrate the performance difference between Software Switching and Hardware Switching. The switch resorts to Software Switching once the cache dedicated to Hardware Switching has reached its maximum capacity.

Hardware Switching uses the TCAM cache to perform the switching at a very high speed, avoiding the involvement of the supervisor - making the process of packet switching very fast. The details of how Hardware Switching works has already been discussed in details, and the results in Experiment 1, quantify the amount of time it takes to do Software Switching. When the destination address of a packet is available in the cache dedicated for Hardware Switching, the switch is able to make a forwarding decision, without relying on the Supervisor.

Software Switching is much slower than Hardware Switching, and is very costly in terms of performance. When a packet arrives, the cache dedicated to Hardware Switching is queried first, if there is a miss, the header information is sent to the Supervisor. The Supervisor then performs a lookup in the cache allocated for Software Switching, if there is a hit, a packet re-write occurs and the packet is forwarded. If on the other hand there is a cache miss, then the Supervisor sends out an ARP request - similar to a miss in the cache when performing Hardware Switching. The difference between Software Switching and Hardware Switching is, when the capacity of Hardware Switching has been reached, all the ARP replies that get processed from that point onward, is stored in the cache dedicated

to Software Switching. A diagram describing the operation of the switch when Software Switching is performed, and a miss occurs is shown below:

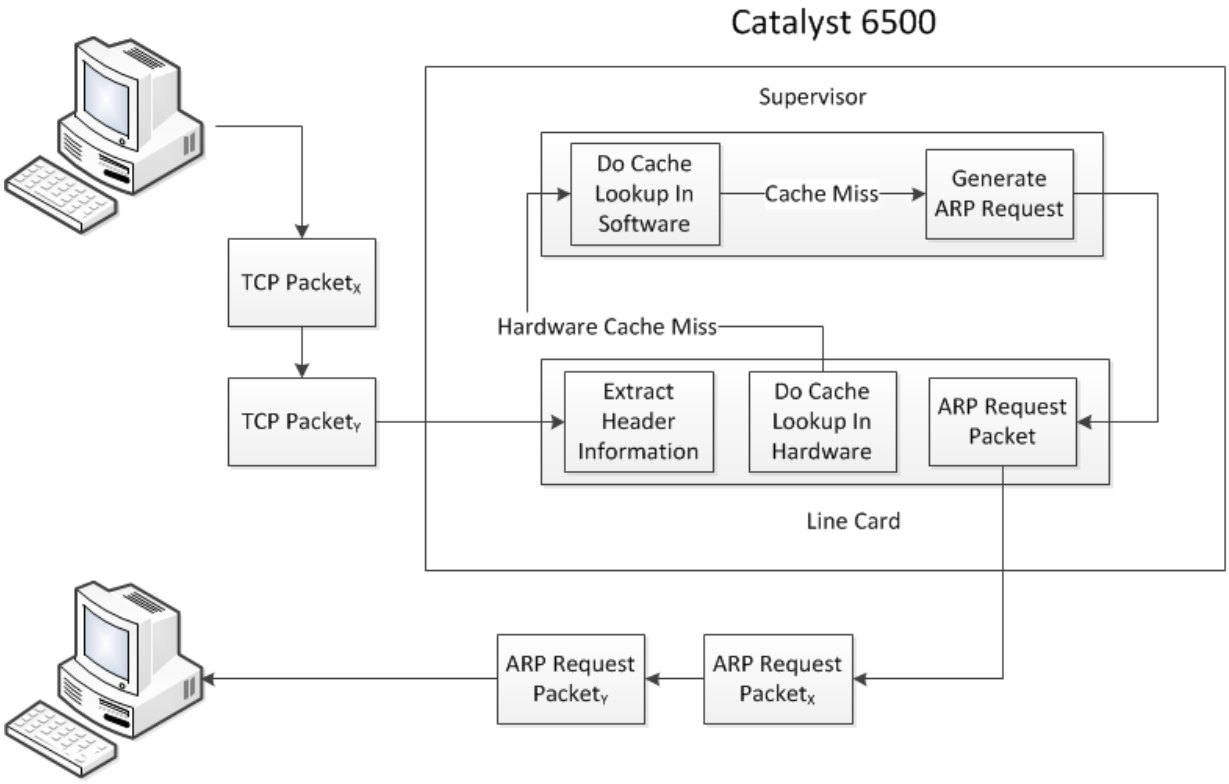

Figure 4.21: Software Switching for a packet with a destination address that does not have an entry in the cache allocated for Software Switching.

A diagram describing the operation of the switch when Software Switching is performed, and there is a hit in the cache dedicated to Software Switching:

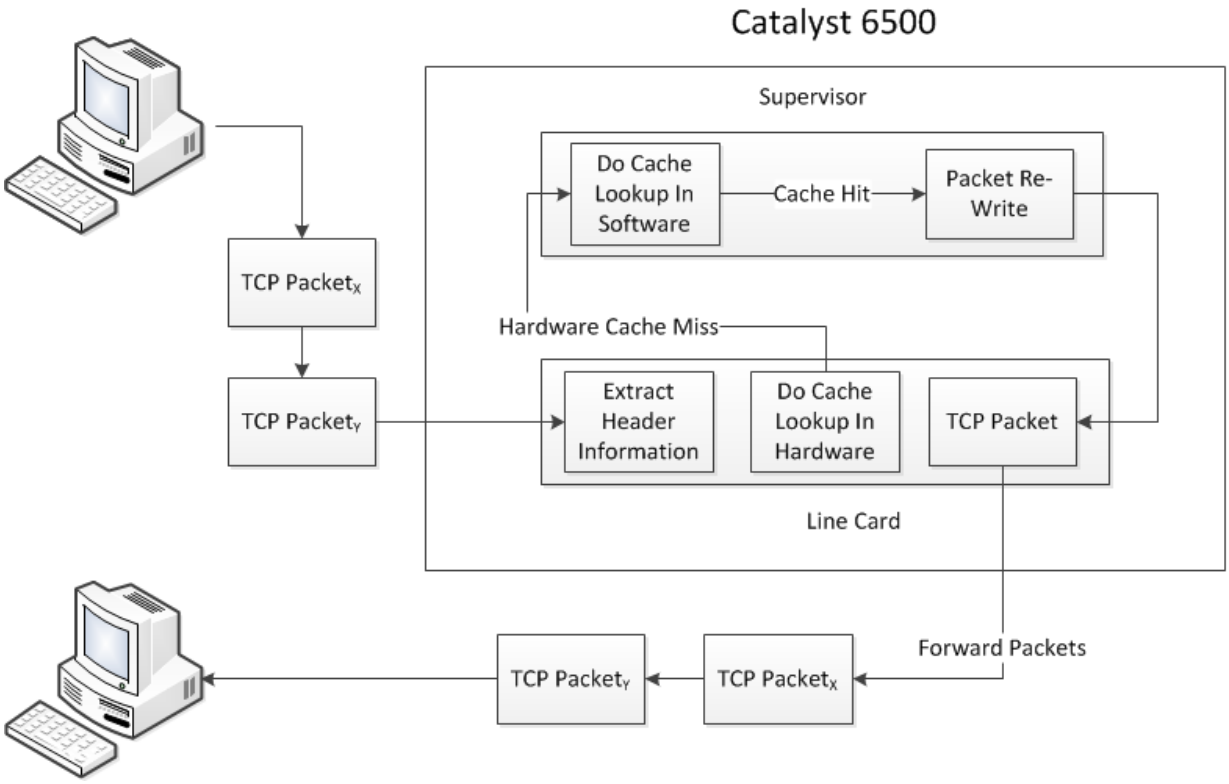

Figure 4.22: Software Switching for a packet with a destination address that has an entry in the cache allocated for software switching.

Since the switch allows the user to set a limit on the number of entries dedicated to Hardware Switching in the TCAM, setting a low limit on the cache allowed us to easily see the effect of going from Hardware Switching to Software Switching. For the purposes of this experiment, the limit on the number of entries for Hardware Switching has been set to 1024 entries. This experiment involved sending TCP packets with destination IP addresses that are not in the cache to fill it up. The number of packets was three times the capacity of the cache dedicated to Hardware Switching, in order to fill it up, and spill over to the cache dedicated to Software Switching. The next step was to send the same set of packets at the maximum rate, and capture the packets at the output. The graph below shows the inter-packet delay of the captured TCP packets:

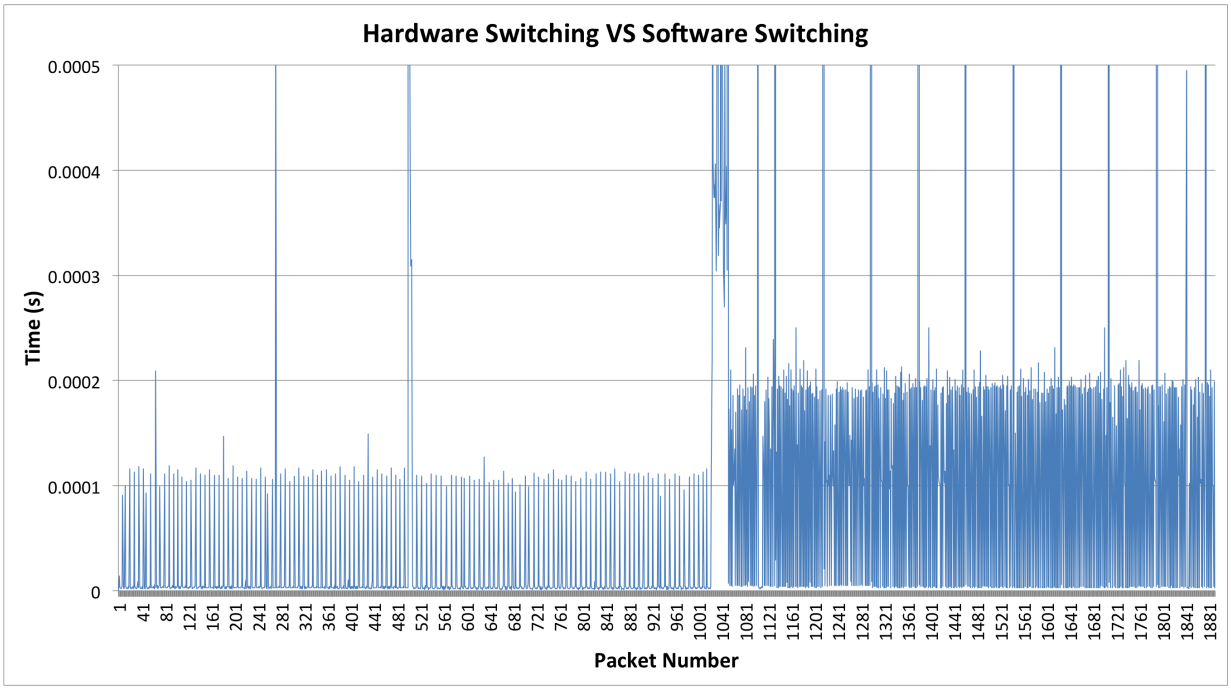

Figure 4.23: Inter-packet delay for switching between Hardware Switching and Software Switching. The switch is configured with a hardware limit of 1024 entries.

For the first portion of the data, upto the limit set for Hardware Switching, the inter-packet delay is in the range of $1\mu s$ to $101\mu s$. After the first 1024 packets, there is a clear jump in the inter-packet delay time. The jump in the delay time is approximately double, which is a fairly significant amount, and thus problematic for the switch's performance.

The statistics for this data is split into two groups, the Hardware Switching statics which includes the first 1024 packets, and Software Switching which includes packets starting at 1025 until the end of the data set.

Statistic	Value (s)
Max Value	84.53 x 10^{-3}
Min Value	0
Average	101.99 x 10^{-6}
Standard Deviation	2.64 x 10^{-3}
Variance	6.98 x 10^{-6}

Table 4.16: Statistics of the inter-packet delay for the first 1024 packets, where Hardware Switching was being performed.

Statistic	Value (s)
Max Value	82.22 x 10^{-3}
Min Value	2 x 10^{-6}
Average	1.12 x 10^{-3}
Standard Deviation	8.95 x 10^{-3}
Variance	8.01 x 10^{-5}s

Table 4.17: Statistics of the inter-packet delay for the packets starting at 1025, till the end of the data set, where Software Switching is being performed.

We can see from the given statistics, that on average, Hardware Switching is in the order of hundreds of microseconds while Software Switching is in the order of ones of milliseconds - a significant difference. The variance and standard deviation are both low, indicating that the distribution of the values from the mean is small, and that the values in the data set are close to the mean - making the average a good representation of the data. This result of course is not surprising since the whole point of Hardware Switching is to maximize the throughput of the packets coming out of the switch.

Chapter 5

Conclusion and Future Work

This section will conclude the work in this thesis, with a summary of the findings, as well as some future work and improvements.

5.1 Findings

In this thesis, a framework for simulating a small network with the switch forwarding packets from one computer to another computer. The framework is designed in such a way so as to minimize any delays that could occur because of the computers involved, which would increase the accuracy of the measurements for the performance of the TCAM. The proposed framework allows the user to fill up the switch's TCAM, providing the basis for investigating a number of properties for the TCAM and switch - such as investigating the penalty associated with a cache miss. The thesis also includes information regarding the challenges associated with implementing the presented framework, and one of many way to overcome them.

Another important set of findings is the quantification of the penalty associated with a miss in the cache while performing Hardware Switching, which on average, is $738\mu s$, compared to approximately an average of $102\mu s$ when there is a cache hit. The penalty associated with a cache miss is divided into two components: the time it takes to realize that there is a miss in the cache and generate an ARP request, and the time it takes to process an ARP reply, add an entry into the cache, and forward the TCP packet. From the results shown, the second component, is the larger portion of the penalty - this makes sense considering the number of tasks that have to completed when processing an ARP reply.

Finally, the quantification of Software Switching versus Hardware Switching. The difference of performance has been measured, and it was confirmed that Software Switching is significantly slower. Software Switching, on average, caused an inter-packet delay of $1.12ms$, while hardware switching was on average $102\mu s$.

5.2 Future Work and Improvements

There are a few things that could be done to improve the results and the robustness of the experiments, as well as more relevant work that could be investigated.

First of all, the rate transmission of packets being fed into the switch should be higher through the use of hardware traffic generators. While the effects of the transmitting computer were significantly minimized over the course of this thesis work, it is of course possible to minimize it further by using hardware traffic generators. This would affect the values measured in the first part of the pnealty, and the performance of packet processing during cache hits.

Another improvement that could be done, and might provide other meaningful results, is the aggregation of two or more switches together. It might also be possible to virtually separate a few ports within the same switch to create an environment that appears to be a different set of clusters of computers or other networking equipment on the network. This could give us information about how the performance could change with the increased number of clusters, or the size of each cluster.

Finally, some future work that could be investigated is a thorough study of cache replacement algorithms, or alternatives to that, which would help reduce the TCAM capacity, without compromising the performance of the switch. Since TCAMs are relatively very expensive, and consume a great deal of power, it is important to optimize the TCAM capacity and ensure that the switch has an optimal TCAM capacity. Since technology is constantly evolving, it is possible that faster, or more efficient cache replacement algorithms, do exist now that could help utilize smaller TCAM capacities.

APPENDICES

Appendix A

Analysis Scripts

A.1 Analyzing Experiment Data

The script used to analyze the capture files to determine the cache miss:

```
import os
from datetime import datetime, time

#Get the file to be analyzed from the user
fileName = raw_input("Enter the file name :")

#Read the tuples for ARP requests from the file
lineReadArpReq = os.popen("tshark -r" + fileName + " -T fields
 -e frame.time_relative -e arp.dst.proto_ipv4 -R 'arp and
 arp.opcode == 1' ")

#Read the tuples for ARP replies from the file
lineReadArpReply = os.popen("tshark -r" + fileName + " -T fields
 -e arp.src.proto_ipv4 -e frame.time_relative -R 'arp and
 arp.opcode == 2' ")
```

```
#Read the tuples for TCP packets from the file
lineReadIpPacket = os.popen("tshark -r" + fileName + " -T fields
 -e ip.dst -e frame.time_relative  -R ip")

#Store the tuples in seperate lists
list_of_arp_requests = lineReadArpReq.readlines()
list_of_arp_replies = lineReadArpReply.readlines()
list_of_ip_pkts = lineReadIpPacket.readlines()

counter = 0

#Create a file to store the delta t of ARP requests
f = open (fileName + '_arp_req_analyze.csv','w')
f.write("This should give us the time it takes the switch to
 receive a packet, and realize there is no entry in the cache \n")

#Iterate through the list of ARP requests and calculate delta t
while (counter < (size_of_list_of_arp_requests-1)):
    parsed_line = list_of_arp_requests[counter].split()
    parsed_line2 = list_of_arp_requests[counter+1].split()
    f.write(float(parsed_line2[0]) - float(parsed_line[0])
    counter = counter + 1

f.close()

#Create tuples of source IP addresses of the ARP replies,
 arp.src.proto_ipv4, and the time of capture (time the
  ARP reply was transmitted), frame.time_relative

list_of_arp_replies_tuples = []

for i in list_of_arp_replies:
    parsed_line = i.split()
    info_tuple = (parsed_line[0].strip(), parsed_line[1].strip())
    list_of_arp_replies_tuples.append(info_tuple)
```

```
#Convert the list of tuples to a dictionary,
  keyed by arp.src.proto_ipv4, with value
  frame.time_relative
  dictionary_of_arp_replies =
  dict(list_of_arp_replies_tuples)

#Create file to store the delta t of arp replies
 and time a TCP packet is received
x = open (fileName + '_arp_rep_ip_analyze.csv','w')

#Loop through the ip readlines, and check the key
 for the time of the arp reply
for i in list_of_ip_pkts:
    parsed_line = i.split()
    t2 = float(parsed_line[1])
    if (parsed_line[0] in dictionary_of_arp_replies):
        t1 = float(dictionary_of_arp_replies[parsed_line[0]])
        delta_t = t2-t1
        x.write (parsed_line[0] + ',' + str(delta_t) + '\n')

x.close()
```

The script used to analyze the capture files to determine the inter-packet delay for TCP packets with a hit in the cache:

```
import os
from datetime import datetime, time

#Get the file to be analyzed from the user
fileName = raw_input("Enter the file name :")
```

```
#Get the tuples of information from the specified file
lineRead = os.popen("tshark -r " + fileName + " -T
 fields -e ip.src -e ip.dst -e tcp.srcport -e tcp.dstport
 -e frame.time_relative -e ip.len -R tcp")

#Create a file to write the results into
f = open (fileName + '_inter_packet_delay_analysis.csv','w')

delta_t = 0.0
counter = 0
second_one_done = False
array_of_lines_read = lineRead.readlines()

#Iterate through the tuples of TCP packet info, and calculate delta t
while counter < len(array_of_lines_read)
    stringArray = array_of_lines_read[counter]
    stringArrayNextOne = array_of_lines_read[counter+1].split()
    delta_t = stringArrayNextOne[4] - float(stringArray[4])
    f.write(stringArray[0] + ',' + stringArray[1] + ',' + stringArray[2] +
     ',' + stringArray[3] + ',' + str(delta_t) +'\n')
    counter += 1

f.close()
```

A.2 Sending and Receiving Packets Scripts

Script used to generate destination IP addresses

```
import os

#Create a file to write the destination IP addresses to
ip_addresses_writer = open('dst_ip_addresses_eth0.txt','w')
```

```
#The first two octets of the destination IP address
ip_address_prefix = '1.100.'

#The third and fourth octets
third_octet = 0
fourth_octet = 2

#This will generate packets from 1.100.0.2 to 1.100.255.253
while (third_octet < 255):
    while (fourth_octet < 253):
       ip_addresses_writer.write(ip_address_prefix+
        str(third_octet)+'.'+str(fourth_octet)+'\n')
       fourth_octet += 1
    fourth_octet = 2
    third_octet +=1
```

The Script used to generate pre-built TCP packets:

```
import os
import socket
from scapy.all import *

#Create and bind the socket to interface eth0
s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
s.bind(("eth0",0))

#Read the file containing the IP addresses
read_file = open('dst_ip_addresses_eth0.txt','r')
#Create a file to store the built packets
write_file = open('built_pkts_eth0.txt','w')

list_of_ips = read_file.readlines()
```

```
for i in list_of_ips:
    pkt = Ether()/IP(dst='%s'%i.strip(),src='1.2.3.4')/TCP()
    write_file.write(pkt.build()+'\n')

read_file.close()
write_file.close()
```

The script used to transmit TCP packets:

```
import os
import socket
from scapy.all import *

#Create and bind the socket - required for transmitting packets
s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
s.bind(("eth0",0))

#Read the file containing the IP addresses
read_file = open('built_pkts_eth0.txt','r')
#Store the built packets in a list
built_pkts = read_file.readlines()

for i in built_pkts:
s.send(i.strip())

read_file.close()
```

The script used to transmit TCP packets using the built in functions:

```
import os
import time
from scapy.all import *
```

```
#grab all the file names
dst_IP_addresses= open('generated_ip_addresses.txt','r')

for dstIP in dst_IP_addresses.readlines():
    generated_packet = IP()/TCP()
    generated_packet.src = '20.0.0.50' #Source IP Address
    generated_packet.dst = dstIP.strip() #Destination IP Address
    generated_packet.sport = 10 #Source TCP port
    generated_packet.dport = 20 #Destination TCP port
    send(generated_packet, verbose =0) #Send generated packet

dst_IP_addresses.close()
```

The script used to generate ARP packets

```
import os
from scapy.all import *
import socket

s = socket.socket(socket.PF_PACKET, socket.SOCK_RAW)
s.bind(("eth0",0x806))

read_file = open('arp_dst_ips.txt','r')
read_built_pkts = open('arp_built_pkts.txt','r')

def arp_monitor_callback(pkt):
   global dict_of_built_pkts
   if ARP in pkt and pkt[ARP].op in (1,2) and
     pkt.pdst in dict_of_built_pkts:
       s.send(dict_of_built_pkts[pkt.pdst])

list_of_dst_ips = read_file.readlines()
list_of_built_pkts = read_built_pkts.readlines()
list_of_pkt_tuples = []
```

```
counter = 0

for i in list_of_dst_ips:
    pkt = list_of_built_pkts[counter]
    pkt_tuple = (i.strip(), pkt)
    list_of_pkt_tuples.append(pkt_tuple)
    counter += 1

#Once we have the list of tuples done, we can build a
 dictionary of that list
dict_of_built_pkts = dict(list_of_pkt_tuples)
print 'Finished preprocessing.. Start transmitting..'
sniff(prn=arp_monitor_callback, filter="arp", store=0)

read_file.close()
read_built_pkts.close()
```

A.3 Analyzing Data From CAIDA

The script used to characterize data from CAIDA:

```
import os
from datetime import datetime, time

#Define the class which will hold the ip_info pairs
class ipInfo (object):
"""This is my ip class"""
src_addr = ''
dst_addr = ''
tcp_src_port = ''
tcp_dst_port = ''
```

```
frame_time_FF = []
packets_per_FF = 0
flow_duration_FF = 0 #This is the total sum of frame_time_FF
avg_time_delay_FF = 0 #This is the sum of flow_duration_ff/packets_per_FF
total_num_bytes_per_FF = 0

def __init__(self, src_addrInit, dst_addrInit, tcpSrcPortInit,
tcpDstPortInit, frameRelativeTimeForwardInit, bytesPerFlowInit, countInit):
      self.src_addr = src_addrInit
      self.dst_addr = dst_addrInit
      self.tcp_src_port = tcpSrcPortInit
      self.tcp_dst_port = tcpDstPortInit
      self.frame_time_FF = list()
      self.frame_time_FF.append(frameRelativeTimeForwardInit)
      self.packets_per_FF = countInit
      self.total_num_bytes_per_FF = bytesPerFlowInit


#Keep track of the total number of packets
total_num_of_packets = 0

fileName = raw_input("Enter the file name :")
print "Opening the file now.."
lineRead = os.popen("tshark -r " + fileName + "
-T fields -e ip.src -e ip.dst -e tcp.srcport -e tcp.dstport
-e frame.time_relative -e ip.len -R tcp")

#Define the array to keep track of order
array_of_keys_order = []
dict_of_ip_infos = {}

for i in lineRead.readlines():
   print "I read:",i,
   #Split the read line and check to see if it has data
   stringArray = i.split()
```

```
   if(len(stringArray) < 4 ):
      print "<Empty line>"
   else:
      dict_key = stringArray[0] + ',' + stringArray[1] + ','
      + stringArray[2] + ',' + stringArray[3]

      dict_value =  ipInfo(stringArray[0],stringArray[1],stringArray[2],
      stringArray[3],float(stringArray[4]), int(stringArray[5]),1)

      #check to see if the entry is in the dictionary
      if (dict_key in dict_of_ip_infos):
      dict_of_ip_infos[dict_key].packets_per_FF += 1
      #print str(dict_value.frame_time_FF[0]) + '\n'
      dict_of_ip_infos[dict_key].
        frame_time_FF.append(dict_value.frame_time_FF[0])
      sumOfDeltaTimes = sum(dict_of_ip_infos[dict_key].frame_time_FF)
      dict_of_ip_infos[dict_key].avg_time_delay_FF = sumOfDeltaTimes/
        dict_of_ip_infos[dict_key].packets_per_FF
      dict_of_ip_infos[dict_key].total_num_bytes_per_FF +=
        dict_value.total_num_bytes_per_FF
        total_num_of_packets += 1
    else:
        dict_of_ip_infos[dict_key] = dict_value
        total_num_of_packets += 1
        array_of_keys_order.append(dict_key)
del(stringArray)

flow_num_counter = 1
f = open (fileName + '.csv','w')
f.write ("Source_ip,destination_ip,tcp_port_src,tcp_port_dst,flow_number,
 packetsPerFlowForward,avg_time_delay_FF,flow_duration_forward,
 flowRateForward,inter_packet_delay\n")
```

```
interpacket_delay = 0.0
key_counter = 0
second_time = False

for keyInDictionary in array_of_keys_order:
  if (second_time == False and key_counter == 1):
    key_counter = 0
    second_time = True
    dict_of_ip_infos[keyInDictionary].flow_duration_FF =
     sum(dict_of_ip_infos[keyInDictionary].frame_time_FF)

   if (dict_of_ip_infos[keyInDictionary].flow_duration_FF > 0):
      flow_rate_FF = dict_of_ip_infos[keyInDictionary].total_num_bytes_per_FF/
       dict_of_ip_infos[keyInDictionary].flow_duration_FF
   else:
       flow_rate_FF = 0

    index_of_delay_calc = array_of_keys_order[key_counter]
    interpacket_delay = dict_of_ip_infos[keyInDictionary].frame_time_FF[0] -
      dict_of_ip_infos[index_of_delay_calc].frame_time_FF[0]

    str_to_write = "%s,%s,%s,%s,%s,%s,%s,%s,%s,%s\n" %
     (dict_of_ip_infos[keyInDictionary].src_addr,
     dict_of_ip_infos[keyInDictionary].dst_addr,
      dict_of_ip_infos[keyInDictionary].tcp_src_port,
      dict_of_ip_infos[keyInDictionary].tcp_dst_port,
      str(flow_num_counter),
      str(dict_of_ip_infos[keyInDictionary].packets_per_FF),
      str(dict_of_ip_infos[keyInDictionary].avg_time_delay_FF),
      str(dict_of_ip_infos[keyInDictionary].flow_duration_FF),
      str(flow_rate_FF), str(interpacket_delay))

f.write(str_to_write)
flow_num_counter += 1
key_counter += 1
```

```
endTime = datetime.now()

f.write ("The total number of forward tcp packets "
 + str(total_num_of_packets) + "\n")
f.close()
lineRead.close()
del(dict_of_ip_infos)
```

Appendix B

Devices' Configuration

B.1 Switch Configuration

```
Building configuration...

Current configuration : 5019 bytes
!
version 12.2
service timestamps debug uptime
service timestamps log uptime
service password-encryption
service counters max age 10
!
hostname Router
!
boot-start-marker
boot system flash disk1:s72033-adventerprise_wan-mz.122-33.SXJ1.bin
boot-end-marker
!
```

```
security passwords min-length 1
no logging console
enable password 7 104F0B1D001B130705
!
no aaa new-model
!
!
!
no ip domain-lookup
mls netflow interface
mls rate-limit unicast cef receive 10000 100
mls cef error action freeze
mls cef maximum-routes ip 1
!
!
!
!
!
!
!
!
spanning-tree mode pvst
no spanning-tree optimize bpdu transmission
diagnostic bootup level minimal
!
redundancy
 main-cpu
  auto-sync running-config
 mode sso
!
vlan internal allocation policy ascending
vlan access-log ratelimit 2000
!
!
!
```

```
interface GigabitEthernet6/1
 no ip address
 shutdown
!
interface GigabitEthernet6/2
 no ip address
 shutdown
!
interface GigabitEthernet8/1
 ip address 100.2.0.1 255.255.0.0 secondary
 ip address 100.1.0.1 255.255.0.0 secondary
 ip address 1.18.173.1 255.255.255.0
 no ip redirects
!
interface GigabitEthernet8/2
 no ip address
!
interface GigabitEthernet8/3
 ip address 1.18.174.1 255.255.255.0
!
interface GigabitEthernet8/4
 no ip address
!
interface GigabitEthernet8/5
 ip address 1.18.175.1 255.255.255.0
!
interface GigabitEthernet8/6
 no ip address
!
interface GigabitEthernet8/7
 ip address 30.0.0.1 255.255.255.0
!
```

```
interface GigabitEthernet8/8
 no ip address
!
interface GigabitEthernet8/9
 no ip address
!
interface GigabitEthernet8/10
 no ip address
!
interface GigabitEthernet8/11
 no ip address
!
interface GigabitEthernet8/12
 no ip address
!
interface GigabitEthernet8/13
 no ip address
!
interface GigabitEthernet8/14
 no ip address
!
interface GigabitEthernet8/15
 no ip address
!
interface GigabitEthernet8/16
 no ip address
!
interface GigabitEthernet8/17
 no ip address
!
interface GigabitEthernet8/18
 no ip address
!
```

```
interface GigabitEthernet8/19
 no ip address
!
interface GigabitEthernet8/20
 no ip address
!
interface GigabitEthernet8/21
 no ip address
!
interface GigabitEthernet8/22
 no ip address
!
interface GigabitEthernet8/23
 no ip address
!
interface GigabitEthernet8/24
 no ip address
!
interface GigabitEthernet8/25
 ip address 100.8.0.1 255.255.0.0 secondary
 ip address 100.9.0.1 255.255.0.0 secondary
 ip address 100.10.0.1 255.255.0.0 secondary
 ip address 100.11.0.1 255.255.0.0 secondary
ip address 100.12.0.1 255.255.0.0 secondary
 ip address 100.13.0.1 255.255.0.0 secondary
 ip address 100.14.0.1 255.255.0.0 secondary
 ip address 100.15.0.1 255.255.0.0 secondary
 ip address 100.16.0.1 255.255.0.0 secondary
 ip address 100.17.0.1 255.255.0.0 secondary
 ip address 100.18.0.1 255.255.0.0 secondary
 ip address 100.19.0.1 255.255.0.0 secondary
 ip address 100.20.0.1 255.255.0.0 secondary
 ip address 100.21.0.1 255.255.0.0 secondary
```

```
 ip address 100.22.0.1 255.255.0.0 secondary
 ip address 100.23.0.1 255.255.0.0 secondary
 ip address 100.24.0.1 255.255.0.0 secondary
 ip address 100.25.0.1 255.255.0.0 secondary
 ip address 100.26.0.1 255.255.0.0 secondary
 ip address 100.27.0.1 255.255.0.0 secondary
 ip address 100.28.0.1 255.255.0.0 secondary
 ip address 100.29.0.1 255.255.0.0 secondary
 ip address 100.30.0.1 255.255.0.0 secondary
 ip address 100.31.0.1 255.255.0.0 secondary
 ip address 100.32.0.1 255.255.0.0 secondary
 ip address 100.33.0.1 255.255.0.0 secondary
 ip address 100.34.0.1 255.255.0.0 secondary
ip address 199.255.253.1 255.255.255.0
 no ip redirects
!
interface GigabitEthernet8/26
 no ip address
!
interface GigabitEthernet8/27
 ip address 199.255.254.1 255.255.255.0
 no ip redirects
!
interface GigabitEthernet8/28
 no ip address
!
interface GigabitEthernet8/29
 ip address 199.255.255.1 255.255.255.0
 no ip redirects
!
interface GigabitEthernet8/30
 no ip address
 shutdown
!
```

```
interface GigabitEthernet8/31
 no ip address
!
interface GigabitEthernet8/32
 no ip address
!
interface GigabitEthernet8/33
 no ip address
!
interface GigabitEthernet8/34
 no ip address
!
interface GigabitEthernet8/35
 no ip address
!
interface GigabitEthernet8/36
 no ip address
!
interface GigabitEthernet8/37
 no ip address
!
interface GigabitEthernet8/38
 no ip address
!
interface GigabitEthernet8/39
!
interface GigabitEthernet8/40
 no ip address
!
interface GigabitEthernet8/41
 no ip address
!
```

```
interface GigabitEthernet8/42
 no ip address
!
interface GigabitEthernet8/43
 no ip address
!
interface GigabitEthernet8/44
 no ip address
 shutdown
!
interface GigabitEthernet8/45
 no ip address
!
interface GigabitEthernet8/46
 no ip address
shutdown
!
interface GigabitEthernet8/47
 no ip address
 shutdown
!
interface GigabitEthernet8/48
 no ip address
!
interface Vlan1
 no ip address
 shutdown
!
router rip
 network 0.0.0.0
!
ip classless
ip forward-protocol nd
!
!
```

```
no ip http server
!
!
route-map test1 permit 10
 set ip next-hop 199.255.255.2
!
!
!
control-plane
!
!
dial-peer cor custom
!
!
!
!
line con 0
line vty 0 4
 password 7 121A04034400331428B3837
 login
!
!
end
```

B.2 Network Configuration File For Transmitter Computer (Computer 1)

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
```

```
address 1.18.173.2
netmask 255.255.255.0
gateway 1.18.173.1

auto eth1
iface eth1 inet static
address 1.18.174.2
netmask 255.255.255.0

auto eth4
iface eth4 inet static
address 1.18.175.2
netmask 255.255.255.0


up route add -net 100.1.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.2.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.8.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.9.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.10.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.11.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.12.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.13.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.14.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.15.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.16.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.17.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.18.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.20.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.21.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.22.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.23.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.24.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.25.0.0 netmask 255.255.0.0 dev eth0
```

```
up route add -net 100.26.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.27.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.28.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.29.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.30.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.31.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.32.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.33.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.34.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.3.0.0 netmask 255.255.0.0 dev eth1
up route add -net 100.4.0.0 netmask 255.255.0.0 dev eth1
up route add -net 100.5.0.0 netmask 255.255.0.0 dev eth1
up route add -net 100.6.0.0 netmask 255.255.0.0 dev eth1
up route add -net 199.255.253.0 netmask 255.255.255.0 dev eth0
up route add -net 199.255.254.0 netmask 255.255.255.0 dev eth1
up route add -net 199.255.255.0 netmask 255.255.255.0 dev eth4
```

B.3 Network Configuration File For Receiver Computer (Computer 2)

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
address 199.255.253.2
netmask 255.255.255.0

auto eth1
iface eth1 inet static
address 199.255.254.2
netmask 255.255.255.0
```

```
auto eth2
iface eth2 inet static
address 199.255.255.2
netmask 255.255.255.0

up route add -net 100.1.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.2.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.8.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.9.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.10.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.11.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.12.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.13.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.14.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.15.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.16.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.17.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.18.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.19.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.20.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.21.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.22.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.23.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.24.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.25.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.26.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.27.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.28.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.29.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.30.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.31.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.32.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.33.0.0 netmask 255.255.0.0 dev eth0
up route add -net 100.34.0.0 netmask 255.255.0.0 dev eth0
```

```
up route add -net 100.3.0.0 netmask 255.255.0.0 dev eth1
up route add -net 100.4.0.0 netmask 255.255.0.0 dev eth1
up route add -net 100.5.0.0 netmask 255.255.0.0 dev eth1
up route add -net 100.6.0.0 netmask 255.255.0.0 dev eth1
```

References

[1] B. Agrawal and T. Sherwood. Modeling tcam power for next generation network devices. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 120–129. IEEE, 2006.

[2] N. Brownlee, C. Mills, and G. Ruth. Traffic flow measurement: Architecture. *Traffic*, 1999.

[3] CAIDA. Internet traffic data. `http://www.caida.org/data/passive/passive_2010_dataset.xml`, March 2012.

[4] Haldun Hadimioglu. *High Performance Memory Systems*. Springer, 2003.

[5] V. Carl Hamacher. *Computer Organization And Embedded Systems*. McGraw-Hill, 2012.

[6] J. Handy. *The Cache Memory Book*. Morgan Kaufmann, 1998.

[7] AG Hanlon. Content-addressable and associative memory systems a survey. *Electronic Computers, IEEE Transactions on*, (4):509–521, 1966.

[8] Craig Hunt. *TCP/IP Network Administration*. O'Reilly Media, 2002.

[9] C.R. Meiners. Algorithmic approaches to optimizing tcam-based packet classification. 2009.

[10] N. Mohan and M. Sachdev. Novel ternary storage cells and techniques for leakage reduction in ternary cam. In *SOC Conference, 2006 IEEE International*, pages 311–314. IEEE, 2006.

[11] P. Nicholson. The application of the in-tree knapsack problem to routing prefix caches. 2009.

[12] R. Panigrahy and S. Sharma. Reducing tcam power consumption and increasing throughput. In *High Performance Interconnects, 2002. Proceedings. 10th Symposium on*, pages 107–112. IEEE, 2002.

[13] B. Parhami. Associative memories and processors: An overview and selected bibliography. *Proceedings of the IEEE*, 61(6):722–730, 1973.

[14] K. Sasai and T. Sasai. Content addressable memory, July 23 1991. US Patent 5,034,919.

[15] Nemesis Software. Nemesis. `http://nemesis.sourceforge.net/`, March 2012.

[16] Scapy Software. Scapy. `http://www.secdev.org/projects/scapy/`, March 2012.

[17] Cisco Systems. Cisco 1600 series router architecture. `http://www.cisco.com/en/US/products/hw/routers/ps214/products_tech_note09186a0080094eb4.shtml`, March 2012.

[18] Cisco Systems. Cisco catalyst 6500 series switches. `http://www.cisco.com/en/US/products/hw/switches/ps708/index.html`, March 2012.

[19] Cisco Systems. Layer 3 interface configuration. `http://www.cisco.com/en/US/docs/switches/lan/catalyst6500/ios/12.2SX/configuration/guide/layer3.html`, March 2012.

[20] Cisco Systems. Rip commands. `http://www.cisco.com/en/US/docs/ios/12_2/ip/configuration/guide/1cfrip.html`, March 2012.

[21] K. Thompson, G.J. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *Network, IEEE*, 11(6):10–23, 1997.

[22] C. Williamson. Internet traffic measurement. *Internet Computing, IEEE*, 5(6):70–74, 2001.

[23] Wireshark. Display filter reference: Address resolution protocol. `http://www.wireshark.org/docs/dfref/a/arp.html`, March 2012.

[24] Wireshark. Display filter reference: Transmission control protocol. `http://www.wireshark.org/docs/dfref/t/tcp.html`, March 2012.