A DIFFERENTIATED WEB CACHE PROXY
WITH A SELF-TUNING FUZZY CONTROLLER

by

NADINE DOMINIQUE SUNDQUIST

B.S. Computer Science, University of Colorado at Colorado Springs, 2007

A thesis submitted to the Graduate Faculty of the

University of Colorado at Colorado Springs

in partial fulfillment of the

requirements for the degree of

Masters of Science in Computer Science

Department of Computer Science

2009
This thesis for Masters of Science in Computer Science
degree by
Nadine Dominique Sundquist
has been approved for the
department of Computer Science
by

Xiaobo Zhou, Chair

Edward C. Chow

Charles M. Shub

11/12/05
Date
Sundquist, Nadine Dominique (M.S., Computer Science)

A Differentiated Web Cache Proxy with a Self-Tuning Fuzzy Controller

Thesis directed by Associate Professor Xiaobo Zhou

As the demand increases on the Internet for more dynamic content from devices such as PDAs, phones, and high-end workstations, so does the load on the servers providing services such as e-commerce and news. In order to alleviate the load on these servers, web cache proxies have become more commonly used to guarantee better end-to-end service. This thesis examines the provisioning of differentiated caching services. First, a self-tuning fuzzy controller algorithm is utilized to allocate cache space for each differentiated class of service based on either the response time or the cache hit rate combined with the number of requests dropped per class of service. Second, the cache replacement policy is a combinatorial approach based on a set of parameters such as page size, least recently used, and least frequently used. The output of the cache replacement policy affects the fuzzy controller. The goal is to increase the cache hit rate and to decrease the response time compared to other controllers such as PID (Proportional, Integral, Derivative). The new algorithm performs better than PID in situations where there is less cache space and higher traffic. This resulting behavior is advantageous because most “real world” situations bring limited cache space, compared to what needs to be stored, and higher traffic, resulting from the increasing popularity of the Internet. The implementation of the algorithms in the Squid web cache proxy showed that a fuzzy controller based approach can provide relative service differentiation mainly in terms of the cache hit rate. It also outperforms the default Squid configuration in terms of the end-to-end response time.
# TABLE OF CONTENTS

## INTRODUCTION

<table>
<thead>
<tr>
<th>Background</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>2</td>
</tr>
<tr>
<td>Quality of Service (QoS)</td>
<td>3</td>
</tr>
<tr>
<td>Self-tuning Fuzzy Controllers</td>
<td>4</td>
</tr>
<tr>
<td>Web Cache Proxies</td>
<td>6</td>
</tr>
<tr>
<td>Caching Policies</td>
<td>6</td>
</tr>
<tr>
<td>Benchmarking</td>
<td>9</td>
</tr>
<tr>
<td>Arrangement of the Thesis</td>
<td>11</td>
</tr>
</tbody>
</table>

## RELATED WORK

| Self-tuning Fuzzy Controllers            | 12 |
| Web Cache Proxies                        | 14 |

## PROBLEM FORMULATION AND ALGORITHMS

| Algorithms                               | 18 |
| Simulation Traffic Generation            | 18 |
| Controllers                              | 19 |
| Fuzzy Controller based on Response Times | 20 |
| Fuzzy Controller based on Cache Hit Rates| 22 |
| PID (Proportional, Integral, Derivative) Controller | 24 |
| Cache Replacement                        | 25 |
Least Recently Used (LRU) Cache Replacement ........................................ 25

Optimized Cache Replacement .............................................................. 26

**DESIGN AND IMPLEMENTATION** ....................................................... 28

Phase I – Caching Proxy Simulation ..................................................... 28
Phase II – Squid Proxy Implementation ................................................ 30
Code Modifications .............................................................................. 30
Network Configuration ......................................................................... 33

**PERFORMANCE EVALUATION** .......................................................... 36

Theoretical Simulation ......................................................................... 38
2:1 Ratio Relative Differentiation .......................................................... 39
Relative Differentiation ....................................................................... 47
1998 World Cup Trace File Simulation ............................................... 56
2:1 Ratio Relative Differentiation .......................................................... 57
Relative Differentiation ....................................................................... 64
Squid Web Cache Proxy ........................................................................ 71

**CONCLUSIONS** .................................................................................. 81

**REFERENCES** .................................................................................... 83
TABLES

Table 1 Simulation Parameters .................................................................38
FIGURES

Figure 1 Simulation Design ........................................................................................................29
Figure 2 Squid Web Cache Proxy Test Network ........................................................................34
Figure 3 Fuzzy Controller based on Response Times – 2:1 Theoretical Simulation –
Response Times..........................................................................................................................40
Figure 4 Fuzzy Controller based on the Cache Hit Rates – 2:1 Theoretical Simulation –
Cache Hit Rates..........................................................................................................................41
Figure 5 Fuzzy Controller based on Cache Hit Rates – 2:1 Theoretical Simulation -
Response Times..........................................................................................................................42
Figure 6 Fuzzy Controller based on Cache Hit Rates – 2:1 Theoretical Simulation - Cache
Hit Rates.....................................................................................................................................43
Figure 7 PID Controller – 2:1 Theoretical Simulation - Response Times .........................44
Figure 8 PID Controller – 2:1 Theoretical Simulation - Cache Hit Rates.........................44
Figure 9 Controller Comparison – 2:1 Theoretical Simulation - Response Times ...........46
Figure 10 Controller Comparison – 2:1 Theoretical Simulation - Cache Hits Rates........47
Figure 11 Fuzzy Controller based on Response Times - Theoretical Simulation -
Response Times..........................................................................................................................49
Figure 12 Fuzzy Controller based on Response Times - Theoretical Simulation - Cache
Hit Rates.....................................................................................................................................50
Figure 13 Fuzzy Controller based on Cache Hit Rates - Theoretical Simulation -
Response Times..........................................................................................................................51
Figure 14 Fuzzy Controller based on Cache Hit Rates - Theoretical Simulation - Cache Hit Rates

Figure 15 PID Controller - Theoretical Simulation - Response Times

Figure 16 PID Controller - Theoretical Simulation - Cache Hit Rates

Figure 17 Controller Comparison - Theoretical Simulation - Response Times

Figure 18 Controller Comparisons - Theoretical Simulation - Cache Hit Rates

Figure 19 Fuzzy Controller based on Response Times – 2:1 1998 World Cup Simulation - Response Times

Figure 20 Fuzzy Controller based on Response Times – 2:1 1998 World Cup Simulation - Cache Hit Rates

Figure 21 Fuzzy Controller based on Cache Hit Rates – 2:1 1998 World Cup Simulation - Response Times

Figure 22 Fuzzy Controller based on Cache Hit Rates – 2:1 1998 World Cup Simulation - Cache Hit Rates

Figure 23 PID Controller – 2:1 1998 World Cup Simulation - Response Times

Figure 24 PID Controller – 2:1 1998 World Cup Simulation - Cache Hit Rates

Figure 25 Controller Comparisons – 2:1 1998 World Cup Simulation - Response Times

Figure 26 Controller Comparisons – 2:1 1998 World Cup Simulation - Cache Hit Rates

Figure 27 Fuzzy Controller based on Response Times - 1998 World Cup Simulation - Response Times

Figure 28 Fuzzy Controller based on Response Times - 1998 World Cup Simulation - Cache Hit Rates
Figure 29 Fuzzy Controller based on Cache Hit Rates - 1998 World Cup Simulation - Response Times...........................................................................................................................................66

Figure 30 Fuzzy Controller based on Cache Hit Rates - 1998 World Cup Simulation - Cache Hit Rates................................................................................................................................................................67

Figure 31 PID Controller - 1998 World Cup Simulation - Response Times...............68

Figure 32 PID Controller - 1998 World Cup Simulation - Cache Hit Rates...............69

Figure 33 Controller Comparisons - 1998 World Cup Simulation - Response Times.....70

Figure 34 Controller Comparisons - 1998 World Cup Simulation - Cache Hit Rates.....71

Figure 35 Squid Proxy - Fuzzy Controller based on Response Times - Response Times 73

Figure 36 Squid Proxy - Fuzzy Controller based on Response Times - Cache Hit Rates.74

Figure 37 Fuzzy Controller based on Cache Hit Rate - Response Times .....................75

Figure 38 Squid Proxy - Fuzzy Controller based on Cache Hit Rates - Cache Hits........76

Figure 39 Squid Proxy - Default Squid Configuration - Response Times .................77

Figure 40 Squid Proxy - Default Squid Configuration - Cache Hit Rates.................77

Figure 41 Squid Proxy - Controller Comparisons - Response Times .....................78

Figure 42 Squid Proxy - Controller Comparisons - Cache Hit Rates......................79

Figure 43 Squid Proxy - Controller Comparisons - Cache Hit Rates Class 1 ..........80
CHAPTER 1

INTRODUCTION

In order to better understand the research and results in this thesis, one needs to understand the fundamentals behind the intent of web cache proxies. This chapter introduces the background and scope of the thesis. It then dives into the fundamentals that encompass Quality of Service, self-tuning fuzzy controllers, and web caches. Finally, the arrangement of the thesis is outlined.

Background

As the demand for e-commerce, news services, and other services increase, the load on servers providing these services increases drastically. Three areas where Quality of Service (QoS) can be guaranteed are the server, the network, and the client. The efficiency and speed of servers have increased, but clients are still inhibited by network congestion. By providing an increased QoS, through a higher cache hit rate and lower response time closer to the client through the use of proxies that cache client requests, clients will have a higher assurance of timely service.

Differentiated QoS has become increasingly important as a myriad of devices have saturated the market, varying from PDAs and phones to high-end workstations [1]. Web content has also become more dynamic than the static pages of yesteryear, which requires more bandwidth and an increased number of calls to databases. A miss in a web
proxy cache has become more expensive in terms of response times due to increased bursty traffic and bigger, more dynamic web sites.

Scope

The proposed solution to the problem of increased user-perceived response time is to provision caching resources using proportional differentiation in a self-tuning fuzzy controller on a web cache server. The policy for what websites are stored in the cache (cache replacement policy) will also be modified. These two components will constitute a two-pronged approach.

The first aspect of the approach is to utilize a self-tuning fuzzy controller algorithm to allocate cache space for each differentiated class of service based on the response time or the cache hit rate[2,1]. If the response time increases for a class of service, then the cache size will also increase proportionally. By increasing the cache size, the cache hit rate should also increase. If once class of service is getting more traffic, then it would require more cache space. Currently, web cache proxies, such as Squid, keep the size of the cache static. If a cache is not as highly utilized for a specific class of service, then the cache space is not transferred to the other classes. By keeping the cache static, precious memory resources are not as effective.

The second aspect of the approach is to remove web sites from the cache that are least frequently used, least recently used, and the biggest page size [3]. Currently web proxies, such as Squid, use a policy of least recently used (LRU) to remove items from the cache. This policy may be simple to implement, but it does not take into account that the least recently used item may have been the most frequently used. Due to the
frequency of the page’s use, the page may have more value in staying in the cache than a page that was last accessed. One needs to take into account user behavior. For example, a user may hit Google or a web mail page on a consistent basis, but it may not have been accessed recently because the user has decided to explore other web pages. Under the least recently used policy, those pages may be thrown out of the cache. However, a new approach of taking into account frequency as well would keep the page in the cache.

By combining the best of these approaches, the hope is to decrease the response time from the point of view of the client and to increase the cache hit rate on the web cache proxy. Preliminary research showed that these two approaches work well separately; therefore, combining these two approaches may bring about prime optimization for a cache web proxy.

This approach was implemented in both a simulation and in the Squid web cache proxy.

**Quality of Service (QoS)**

Quality of Service (QoS) is defined to be the measure of end-to-end system performance according to timeliness and bandwidth guarantees. Two strains of QoS exist, including Integrated Service (IntServ) and Differentiated Services (DiffServ). In an IntServ system, a client is able to demand a specific level of service, which makes this system reservation based. Even though this system may sound ideal, some clients will get dropped from service entirely if all the resources of a system are already used up. Dropping clients entirely is unacceptable.
DiffServ provided differentiated services between classes of aggregated flows of traffic, instead of individual flows of traffic. The DiffServ option can also be split into two other components, absolute differentiated service and relative differentiated service. In absolute DiffServ, statistical data is used on the aggregate flows of traffic to guarantee a certain range of performance, which is paramount for streaming applications. Relative DiffServ provides relative QoS between classes. A class that is considered a higher level of service will be guaranteed at least equal or better service than a lower class. With this system, no clients will be dropped. [4]

Maintaining a level of quality of service, especially differentiated QoS, can be difficult especially in places such as 802.11 wireless networks [5]. In [6], the authors showed how they tried to achieve absolute differentiable service for multimedia services, while other traffic received relative differentiable service. They achieved their differentiation by letting the number of idle windows determine the size of the contention window.

The focus of this thesis is on relative differentiated QoS in web cache proxies. DiffServ is highly effective for the average web application. Therefore, it would be quite advantageous in the implementation of a web cache proxy, where the cached items are simple web pages or pieces of web applications.

Self-tuning Fuzzy Controllers

For well over a decade, self-tuning fuzzy controllers have existed in either physical or logical form to fine tune settings based on a feedback control mechanism. Self-tuning fuzzy controllers have varying applications, anywhere from the seismic
resistance of buildings (more hardware oriented) to bounding performance metrics (more software centric).

For example, Lama and Zhou in [7] utilized a fuzzy controller to bound the 90\textsuperscript{th} percentile delay of requests flowing through a multi-tier architecture. Their goal was to provision servers to guarantee a lower end-to-end delay.

In 2009, a fuzzy controller was presented in [8] where relative differentiated quality of service was achieved by implementing a fuzzy algorithm based on the JOBS (Joint Buffer Management and Scheduling) algorithm. The input into the fuzzy controller was the delay and the packet loss. The authors were able to achieve the goal of decreasing the packet loss rate of the higher classes of service.

Variations of PID (Proportional, Integral, Derivative) fuzzy controllers have been the most common way of trying to solve many different types of problems. PID controllers are quite difficult to tune, and they usually adapt to linear circumstances the best. One recent example of research in optimizing PID controller is when the authors of [9] came up with a PID controller that could adapt to non-linear environmental factors, but would as a rule follow a linear behavior.

Any problem that can take advantage of inputs to positively affect the overall environment can use a fuzzy controller. Within the last couple of years, logical fuzzy controllers have been frequently used to allocate resources based on negative feedback. For example, if the response time of a server lowers, then it should be allocated more resources.
Web Cache Proxies

Web caches reduce the use of network bandwidth, server load, and response time by caching web components such as HTML pages and images. Web cache proxies have become increasingly important as the use of the Internet increases; therefore, there has been a heavy emphasis on increasing the efficiency of web cache proxies. One such effort is the Squid web cache proxy, where its open source has allowed programmers to provide multiple solutions [10]. Research from multiple sources, including Hewlett Packard Laboratories [11], has shown that web cache proxies can significantly reduce the response times up to half the time when the proxy is closest to the end user.

In 2007, the authors of [12] applied the theory that web cache proxies could also speed up requests from mobile devices. They were able to show that the overall latency of requests could be reduced for mobile devices using all the major protocols when a web cache proxy was utilized.

In the same vein, the authors of [13] suggested using web cache proxies that are dedicated to P2P (Peer to Peer) traffic. Their research brought them to the conclusion that P2P traffic has many characteristics unlike normal traffic. They found that the classical cache replacement policies such as LRU (Least Recently Used) and Least Frequently Used (LFU) negatively impact the cache hit rates. This research illustrates that web cache proxies in their default state cannot be applied to just any problem. Sometimes a completely different algorithm has to be created to service unique kinds of traffic that reflect varying types of user behavior.

Caching Policies
Caching is highly influenced by cache replacement policies [14,15,16,17]. The more efficient a cache is used, the higher the cache hit rate. A by-product of higher cache hit rate can be a faster response time, even though these two factors are not necessarily proportional.

The simplest cache replacement policy is First-In-First-Out (FIFO), which acts similar to a queue at the super market. Whoever arrived first is the first to leave. The next algorithm is Last-In-First-Out (LIFO), which is a stack where the last item is replaced by an incoming item [14].

The most common replacement algorithm is Least Recently Used (LRU). When an item is accessed in a cache, it is moved to the Most Recently Used (MRU) spot. The item at the tail of the list is removed when cache space becomes limited.

Another popular algorithm for cache replacement is Least Frequently Used (LFU). This algorithm provides the opportunity to expunge unpopular items in the cache. The downfall of this algorithm is that cache pollution occurs where popular items in the distant past are impossible to remove [14].

Yet another cache replacement algorithm is based on size. Larger objects are replaced to make room for many smaller objects. This can be advantageous in web cache proxies because users usually request smaller items more often than larger items.

In the Squid web cache proxy, Least Recently Used (LRU) is defaulted. However, in 1999 a few researchers from Hewlett-Packard laboratories implemented a couple more solutions in the Squid web cache proxy that combined multiple cache replacement algorithms [18].
The first algorithm was Least Frequently Used with Dynamic Aging (LFU-DA). This algorithm combines the strengths of LFU with LRU. Both frequency and time of access are considered.

The second algorithm implemented was Greedy Dual-Size Frequency (GDSF). This algorithm focuses on favoring smaller, popular objects to increase the hit rate. When items are in a tie, then the age of an item is taken into account.

Hewlett-Packard implemented these two algorithms in the Squid web cache proxy using a heap mechanism. Their research and testing using SPEC Web benchmarking resulted in the conclusion that LFU-DA outperformed LRU with a better cache hit rate. Due to the fact that they did not have to touch the objects in the cache as often, these algorithms were less prone to increase CPU cycles and less prone to create thrash in the cache [18].

An example of bringing fuzzy logic to cache replacement policies was brought up in [19], where the authors use fuzzy logic to determine if an item should reside in a long-term cache or in a short-term cache on the client-side proxy. They also determined if an item should be kept in a cache based on last accessed, frequency, size, and access latency using fuzzy logic.

So far, cache replacement algorithms have had the highest focus in increasing performance on web cache proxies. However, in recent years, developing controllers based on user behavior has gained steam.
Benchmarking

This thesis uses three methods of benchmarking. For simulations, theoretical traffic generators and the 1998 World Cup trace files are used. For benchmarking the Squid web cache proxy, a widely used benchmarking tool called Web Polygraph is utilized.

Theoretical Traffic Generator

In a simulated web cache proxy environment, the simplest way to measure performance is to create a randomized traffic generator. For measuring a web cache proxy’s performance, the factors considered in the generator are the number of websites generated, the average web site size, and the average arrival time. By randomizing these factors, a steady stream of traffic can be generated for a simulation. Theoretical generators give more of an indication of how an algorithm will behave under certain load conditions, but it does not necessarily indicate how well an algorithm will behave under unpredictable, real-world conditions.

World Cup 1998

The most common trace files used to rate controller algorithms for the past decade are the 1998 World Cup traffic trace files [20,21]. This traffic during the 1998 World Cup games was captured over a period of several months. Multiple researchers have used these trace files as a baselines for heavy traffic for both web cache proxies and storage caches [22,2]. In [22], the authors used the 1998 World Cup trace files to evaluate storage cache controllers including a linear controller, a gradient controller, a PID controller, and a retrospective controller. One can conclude by using a thorough understanding of both
web cache proxy theory and storage theory that these two sometimes share algorithms. Therefore, they both can also share many benchmark procedures.

**Web Polygraph**

In 2001, 10 caches proxy were analyzed using a highly popular benchmarking tool called Web Polygraph [23]. Squid, at the time, performed poorly against other commercial products [24]. With great room for improvement, the open source community took over developing Squid, and it has significantly improved since this last evaluation.

Web Polygraph is a freely available tool used for benchmarking Web intermediaries including web cache proxies and origin server accelerators. Through a bit of fine-tuning, Web Polygraph is able to generate fairly realistic web traffic. This tool consists of two program components, a client and a server. Multiple servers and clients (robots) can be started up. These servers and clients can talk directly, but during benchmarking the clients are told to route through a specified proxy.

Predefined workloads with specific characteristics, such as faster and slower load cycles, have already been created. These files only need to be modified with environment-specific changes such as IP addresses. The following are examples of how the servers and clients are started up:

- polyclt --config workloads\polymix-1.pg --verb_lvl 10 --proxy 192.168.200.2:3128
- polysrv --config workloads\polymix-1.pg --verb_lvl 10

Web Polygraph is able to generate statistics on the benchmarking including hit rate, user response times, and error rates. [23,25,26,27] In [26], the authors were able to
use Web Polygraph to successfully prove that multiple Squid web proxies working together provided better performance than a single web proxy working by itself.

**Arrangement of the Thesis**

This thesis starts off by giving a more detailed account of related work in Quality of Service, self-tuning fuzzy controllers, and web cache proxies. Problem Formulation and Algorithms then continues to define the problem and outline a solution as to how to make a web cache proxy even more efficient in order to achieve the goal of increasing the web cache proxy hit rate and decreasing the response time. This same section outlines the formulas depicting the mathematical theory applied in the implementation with clarifications as to why these algorithms were utilized. In the Design and Implementation section of the paper, the aforementioned algorithms are applied to different implementations. The first implementation exhibits a theoretical simulation written in the Java programming language. The second design displays how the algorithms were implemented in a Squid web cache proxy. In Performance Evaluation, the first subsection summarizes the results of the theoretical simulations while the second subsection shows the results for benchmarking the Squid cache proxy after modifications. Finally, the last section ties together the results with a succinct Conclusion.
CHAPTER 2

RELATED WORK

The work of this thesis bases itself upon previous work in the fields of fuzzy controllers, cache replacement algorithms, and web cache proxies. The conference papers on controllers had the greatest impact on my research.

Self-tuning Fuzzy Controllers

In [22], the authors used what they called a per-class feedback controller to improve a shared storage cache. They compared different kinds of per-class feedback controllers including a linear controller, a gradient-based controller, a PID (Proportional, Integral, Derivative) controller, and a retrospective controller. They had introduced the retrospective controller and found that it was comparable to the PID controller in achieving a specified cache hit ratio. Even though this paper is primarily focused on storage caches, the algorithms used are also applicable to web cache proxies.

The mathematical model of the PID controller in this paper was the primary source for the PID controller in the comparison of my new fuzzy controller to an established algorithm. The PID controller is highly sensitive and difficult to tune, but the authors of [22] were able to prove that the cache hit rate does not oscillate as much as some other controllers such as the gradient controller. This led me to believe that it would make this algorithm ideal for comparisons against my new algorithms. The one weakness in [22] is that differentiated service with multiple classes was not achieved cleanly. The oscillations in service for the classes of service were extreme and not very predictable.
The conventional PID controller has existed for well over half a century; they have been successfully applied in industrial systems. However, their downfall is that they are not as effective in non-linear systems as pointed out by [28]. These authors decided to use a fuzzy PID controller to manage congestion control. The PID algorithm has three constants that have to be fine-tuned, and their approach was to use a set of rules (fuzzy logic) to determine the values of each these constants.

For the last ten years, researchers have been experimenting with fuzzy PID controllers to find the best way to fine-tune the controller [29,30]. Most of these fuzzy PID controllers have been applied to industrial situations such as controlling solar radiation or oil flow. The primary way of providing feedback on how to tune the fuzzy controller is to use a series of if-else statements that determines the values of the three constants.

Even though PID controllers are widely used in industry, their downfall is that they are difficult to tune, and they do not always adapt well to situations where input to the feedback control mechanism is sporadic and unpredictable. In Internet traffic, this is very much the case. A classic PID controller would be unable to cope with the changes, which is why fuzzy PID options have been explored.

Since PID is so difficult to tune, this gives others an incentive to provide alternate solutions to PID that are easier to set up and more responsive to randomized traffic.

In 2006, the authors of [31] explored using a differentiated service controller based on Web caching and Content Distribution (CDN), such as file type. They incorporated different caching algorithms such as LRU and LFU-DA. Their algorithm was original in that they assigned a different cache replacement algorithm to each class of
service. For example class 1 would be using LRU, while class 2 would be using LFU-DA. I did not see that much logic behind this move, but they achieved differentiation between two classes with accuracy and very little oscillation in the QoS.

Their algorithms were based on the assumptions that 60-70% of objects are never accessed again, most objects accessed are small, images are mostly requested followed by HTML, the number of accesses to dynamic objects is minimal, and web access exhibits properties of temporal and spatial locality. I separately came to this same conclusion, and based my algorithms on the same idea by not keeping large items in the cache.

The authors of [31] experienced the same problem that I did in that having more than two classes of service caused quite a bit more overhead that would slow down the server. This is due to the sheer number of calculations required for each class, which is quite a bit more than the classical LRU, where the calculations are minimal.

**Web Cache Proxies**

There were two papers on web cache proxies that had a significant impact on my research. The first approach I found intriguing is primarily based on a paper where the authors implemented a self-tuning fuzzy controller that allocated cache space for each proportionally differentiated class based on a feedback control mechanism. If the response time for a specific class was too high, then that class would be given a higher percentage of the overall cache [1]. Each class was based on the type of device that was accessing the web content. For example, one class would consist of wireless devices such as PDAs and phones, while another class would consist of workstations. I personally think this kind of approach would be advantageous because each kind of device tends to
access different kinds of web pages. A phone or PDA would be more prone to access news sites or e-mail, while a workstation would be more prone to access e-commerce and media.

The authors of paper [1] were successfully able to prove that a self-tuning fuzzy controller on a Squid proxy cache could outperform the standard proxy cache. Latency reduction of the QoS model of Squid was higher than the standard Squid configuration.

The Squid web cache proxy uses Least Recently Used (LRU) to determine which pages are evicted from the cache. The authors of [3,32,33] simulated an approach where pages are evicted based on the following factors:

- size of the page
- frequency of use of the page
- time of the page’s last access

Each page has a probability of being dropped based on these factors. These authors compared their approach to Least Recently Used (LRU), Least Frequently Used (LFU), and Greedy Dual Size (GDS), which was specifically designed for web environments [3,17]. They were able to prove that their fuzzy algorithm for cache eviction was better than any one of these algorithms.

The Squid cache web proxy (originally started through an NSF – National Science Foundation - grant) has emerged as one of the primary proxies due to its wide acceptance (due to no cost) and the ability to modify the source code [10].

Web cache proxies provide several areas for increasing efficiency, and these are only a few areas where people tried to improve the proxy.
CHAPTER 3

PROBLEM FORMULATION AND ALGORITHMS

Today’s devices consist of everything from an iPhone to a high-end server with varying speeds in Internet connections and varying requests in content. As more devices are invented to interact with web pages containing dynamic content, traffic will increase causing higher network congestion and more load on servers. Storage has become denser with more applications sharing the same space than ever before, which means storage needs to respond to more requests driving up response times [22]. In order to alleviate this server load and to decrease the response times of requests, web cache proxies need to have a higher cache hit rate.

By distributing the load across multiple web proxy caches across the Internet, closer to the client, storage services and web servers will be able to service a higher number of clients with a minimal impact on response time [22]. In the real world, such as in the realm of e-commerce and news services, the response time would decrease; thus, customer satisfaction would increase.

In this thesis, the utilization of a bilateral approach decreases the response time from the point of view of the client. The first part of the approach is to utilize a self-tuning fuzzy controller algorithm to provide cache space based on relative proportional differentiation, client perceived latency, cache hit rate, and the rate of cache replacement (removing items from the cache) [2,1].
In this model, there are multiple classes of service. If the response time increases too much for a class of service, then the fuzzy controller will allocate a higher percentage of the cache space to that class. Also, if the cache hit rate decreases too much for a class of service, then more cache space will be allocated to that class. Another factor that increases the size of the cache is if too many items are being replaced in a cache. Too much thrash (high replacement) in a cache throws off the stability of the cache. The self-tuning fuzzy controller can be told to check at certain intervals to make sure that the cache sizes are correct. The fuzzy controller allows the cache to be more responsive to the needs of the users.

The second portion of the approach is to evict items in the cache based on parameters such as the size of the page, when it was most recently accessed, and how frequently the page was accessed [3].

In the mechanism that removes items from the cache, I would have wished to have the opportunity to try adding one more factor to the probability of a page being dropped, other than just the page size, time of access, and frequency of access. I think it would be interesting to have a higher probability of dropping a web page from the cache if it has more dynamic content. One important point that was focused on in [1] was the impact that static HTML versus dynamic web content could have on improving cache performance. My assumption would be that more requests could be serviced in a shorter amount of time if static pages were served up first.

By combining these two approaches, I hope to decrease the response time from the point of view of the client and increase the cache hit rate in the web cache proxy. Both of the components of this approach would be both predictable and controllable. The
approach would be predictable in that higher classes would get better service than lower classes, and it would be controllable in that the parameters of the fuzzy algorithm could be manually changed by an administrator. By making sure that predictability and controllability are met, the algorithm would be considered fair because no client would be starved of requests.

**Algorithms**

A series of algorithms were used for simulation traffic generation, cache size manipulation, algorithm comparison, and cache replacement policies. The algorithms that were new and focused on cache size manipulation were a fuzzy controller based on response times and a fuzzy controller based on cache hit rates. For comparison, a popular algorithm called PID (Proportional, Integral, Derivative) was applied. For cache replacement (removing items from the cache), either LRU (Least Recently Used) or an optimized cache removal policy were used.

**Simulation Traffic Generation**

The simulation required website requests, and these requests can either be theoretically created or pulled from an existing trace file. The theoretical generator created a pool of websites available and generated requests for the simulation.

Before the simulation starts, the user inputs the number of websites in the pool, the maximum website size, and the maximum arrival interval of the requests. The theoretical generator builds a pool of websites for the simulation by iterating through the loop:
foreach (website) {
    generate random website size between 1 and max website size;
}

The theoretical generator randomly picks from this pool of websites and sends the request with following loop:

foreach (request) {
    arrivalInterval = random between 0 and max arrival interval;
    classOfService = random, either 1 or 2;
    send random website request with random arrival interval and random class;
}

This theoretical traffic may not reflect a real-world scenario, but it does provide the ability to randomize behavior within a specified set of parameters. Varying behavior can be ascertained by changing the number of websites, the arrival interval time, and the size of the simulated cache.

**Controllers**

When the website request arrives, the type of controller specified by the user will process the request and determine the size of the cache based on the algorithm. The controllers used were a new fuzzy controller based on response times, a new fuzzy controller based on cache hit rates, and a PID (Proportional, Integral, Derivative) controller for comparison.
**Fuzzy Controller based on Response Times**

This controller changes the size of the cache based on the response time feedback. A ratio between the two caches determines the amount of space assigned to each cache. The three factors that affect the ratio are:

- the response times per class
- the relative differentiation between classes e.g. 2:1 ratio between class 1 and class 2
- the number of requests dropped per class

**Step 1**

The ratio is first calculated using the average response times per class:

- ratio = \( t_1 / t_2 \)

The values are the following:

- \( t_1 = \) average response time of cache 1 (class 1)
- \( t_2 = \) average response time of cache 2 (class 2)

**Step 2**

Next, the ratio takes into account the number of requests that have been dropped per class. By taking this number into account, we are balancing out change that may happen too quickly, which can cause too much turnover in the cache. If a cache is dropping too many requests, then it needs more space. This does make the cache more efficient in correctly allocating space, but it can pull the algorithm away from achieving perfect relative differentiation e.g. a 2:1 ratio between classes. The ratio is affected with one of the following changes:

- if (turnoverRatio > classDifferentiation) { ratio *= turnoverRatio; }


if (turnoverRatio < classDifferentiation) { 
    ratio /= turnoverRatio; 
}

The values are the following:

- turnoverRatio = \frac{\text{requestsDropped}_1 / \text{totalRequests}_1}{\text{requestsDropped}_2 / \text{totalRequests}_2}
- classDifferentiation: if 2:1 ratio, then class differentiation is 2.

**Step 3**

Then, the ratio takes into account the relative differentiation between classes e.g.
a 2:1 ratio. The ratio is affected with the following modification:

- ratio *= classDifferentiation

The values are the following:

- classDifferentiation: if 2:1 ratio, then class differentiation is 2.

**Step 4**

Finally, the ratio is applied to each class’s cache size. Initially, the cache size is set to be equal between all classes, but from then on is affected by the calculated ratio in Steps 1-3. If the newly calculated cache size is greater than the total cache size, then the cache size stays the same. The following shows the ratio applied to the cache sizes:

- if (ratio > 1) { 
  cache1Size *= ratio; 
  cache2Size = totalCacheSize – cache1Size; 
}

- if (ratio < 1) { 
  cache2Size /= ratio; 
  cache1Size = totalCacheSize – cache2Size; 
}

The values are the following:

- ratio = ratio calculated in Steps 1-3
- cache1Size = the current size of cache 1 (class 1)
o cache2Size = the current size of cache 2 (class 2)
o totalCacheSize = the size of the entire cache (cache1Size + cache2Size)

The websites are removed from the cache based on an optimized removal algorithm. A website has a higher probability of being removed if the size of the website is greater than the average website size, the website is least recently used, and the website is least frequently used. Each of these components is considered a penalty against the website for being kept.

**Fuzzy Controller based on Cache Hit Rates**

This controller changes the size of the cache based on the cache hit rate feedback. A ratio between the two caches determines the amount of space assigned to each cache. The three factors that affect the ratio are:

- the cache hit rates per class
- the relative differentiation between classes e.g. 2:1 ratio between class 1 and class 2
- the number of requests dropped per class

**Step 1**

The ratio is first calculated using the average response times per class:

- ratio = h2 / h1

The values are the following:

- h1 = average cache hit rate of cache 1 (class 1)
- h2 = average cache hit rate of cache 2 (class 2)

**Step 2**
The ratio next takes into account the number of requests that have been dropped per class. By taking this number into account, we are balancing out change that may happen too quickly, which can cause too much turnover in the cache. If a cache is dropping too many requests, then it needs more space. This does make the cache more efficient in correctly allocating space, but it can pull the algorithm away from achieving perfect relative differentiation e.g. a 2:1 ratio between classes. The ratio is affected with one of the following changes:

- if (turnoverRatio > classDifferentiation) { ratio *= turnoverRatio; }
- if (turnoverRatio < classDifferentiation) { ratio /= turnoverRatio; }

The values are the following:

- turnoverRatio = (requestsDropped₁ / totalRequests₁) / (requestsDropped₂ / totalRequests₂)
- classDifferentiation: if 2:1 ratio, then class differentiation is 2.

**Step 3**

Then, the ratio takes into account the relative differentiation between classes e.g. a 2:1 ratio. The ratio is affected with the following modification:

- ratio *= classDifferentiation

The values are the following:

- classDifferentiation: if 2:1 ratio, then class differentiation is 2.

**Step 4**

Finally, the ratio is applied to each class’s cache size. Initially, the cache size is set to be equal between all classes, but from then on is affected by the calculated ratio in
Steps 1-3. If the newly calculated cache size is greater than the total cache size, then the cache size stays the same. The following shows the ratio applied to the cache sizes:

- if (ratio > 1) { cache1Size *= ratio; cache2Size = totalCacheSize – cache1Size; }
- if (ratio < 1) { cache2Size /= ratio; cache1Size = totalCacheSize – cache2Size; }

The values are the following:

- ratio = ratio calculated in Steps 1-3
- cache1Size = the current size of cache 1 (class 1)
- cache2Size = the current size of cache 2 (class 2)
- totalCacheSize = the size of the entire cache (cache1Size + cache2Size)

The websites are removed from the cache based on an optimized removal algorithm. A website has a higher probability of being removed if the size of the website is greater than the average website size, the website is least recently used, and the website is least frequently used. Each of these components is considered a penalty against the website for being kept.

**PID (Proportional, Integral, Derivative) Controller**

The controller changes the size of the caches based on the following equation:

$$s_i(n+1) = s_i(0) + K_P e_i(n) + K_I \sum_{j=0}^{n-1} e_i(j) + K_D \Delta e_i(n)$$

In plain English, the values are the following:

- $s_i(0) = \text{starting size for class } i$
- $K_P$ = Proportional constant
- $K_I$ = Integral constant
- $K_D$ = Differential constant
- $e_i = (target\ hit\ rate - current\ hit\ rate)$
- $s_i(n + 1) = new\ cache\ size\ for\ class\ i$
- For these runs, the constants were all set to -10, 10, and 10, respectively.

The $K_P$ portion of the equation allows for immediate changes in the class by taking into account only the current hit rate, which can make the cache size fluctuate quickly. The $K_I$ portion of the equation changes the size of the cache based on how far the current hit rate was from the target hit rate for the last few cache size adjustments. The $K_D$ portion of the equation stabilizes the system by taking into account the entire history of the cache hit rate fluctuations [9,22].

The cache is using a Least Recently Used (LRU) mechanism to remove websites from the cache.

Cache Replacement

Each of the controllers employs a different cache replacement algorithm. The fuzzy controller based on response times and the fuzzy controller based on cache hit rates both utilize an optimized replacement algorithm, while the PID controller utilizes a Least Recently Used (LRU) replacement algorithm.

Least Recently Used (LRU) Cache Replacement
The most common and easiest cache replacement algorithm is Least Recently Used (LRU). Every time an item is accessed, it is put at the top of the queue. When a cache becomes too full, the items at the tail of the queue are removed first. This simple algorithm only takes into account the time of access, and its implementation in any code base is effortless. The often-used Squid web cache proxy defaults to using LRU as its cache replacement policy.

**Optimized Cache Replacement**

Optimized cache replacement has been explored by using combinations of other cache replacement policies. In this thesis, such an optimized cache replacement policy was made use of in combination with the fuzzy logic that an item will not even be stored in a cache if the item is greater than 10% of the size of the cache. Displacing many smaller items in the cache with such a large item can be costly.

Each item in the cache is assigned a specific removal probability based on web site size, the time that the item was last accessed, and the frequency of use of the item. The item with the highest removal probability is the next item to be removed.

**Step 1**

Each item starts with a removal probability of 0. The first step is to take into account the size of the object:

- if (website size > average website size) { removalProbability++; }

**Step 2**

Next, the age of the item is taken into consideration:

- removalProbability += time since last accessed;
Step 3

Then, the frequency of use of the item is included into the probability:

- removalProbability -= access frequency;

Step 4

Finally, the item with the highest probability is removed. The number of items removed for each class (cache) is monitored in order to take it into account in the fuzzification logic of the fuzzy controller.

By taking into account time of use, frequency of use, and size of request, we are able to bind together the positive attributes of these algorithms without falling into the trap of cache pollution. These factors also take into account a user’s behavior when surfing the Internet. Someone may use Google frequently, but may not use it for a couple of hours. If the LRU cache replacement algorithm is used, then Google would be removed from the cache. This optimized algorithm allows an item such as this to be kept in the cache, which allows a higher cache hit rate in the future.
CHAPTER 4
DESIGN AND IMPLEMENTATION

The implementation of the cache proxy algorithms was completed in two phases. The first phase was a caching proxy simulation implemented in Java. The second was an implementation of the algorithms in a Squid web cache proxy.

Phase I – Caching Proxy Simulation

My first implementation was of the web cache proxy simulation in order to prove that the theory behind the thesis functioned as planned. The following are the components of the simulation:

(1) The first component represents the Internet, and it either generates requests or pulls information from existing trace files.
   a. Auto-generation: To make the simulation simple, a pool of websites of varying sizes are created. The Internet generator then randomly picks from the pool of websites at random intervals of time and assigns a random class of service.
   b. Trace files: Each line is read out of the trace file per simulation. In our case, we are using a trace file from the 1998 World Cup website [20].

(2) The second part of the simulation would be the cache. The cache receives requests for either Class 1 or Class 2. If the website does not reside in the cache, a random time unit penalty is added to the response time to represent having to go beyond
the cache. The item is then added to the cache if the item is not greater than 10% of the size of the cache. If the item resides in the cache, then no penalty will be added to the response time.

(3) If the cache gets too full, then items will need to be dropped from the cache. There will be two methods implemented for comparison. The classical LRU (Least Recently Used) and an optimized dropping mechanism have been implemented. The optimized mechanism will take into account least recently used, least frequently used, and page size when dropping an item from the cache.

(4) The third component of this system is the self-tuning fuzzy controller that affects the size of the cache. If the response time gets too large for a class or if the cache hit rate is too low, then that class will be assigned more space.

The simulation was written using Java as a programming language. The implementation was easily able to handle a million requests within seconds.

The elements proving success would be the following:
• higher cache hit rate
• lower client perceived response time

In this simulation, two different fuzzy controllers are tested. The first fuzzy controller based on response times varies the cache size based on the feedback received from the response times. The second fuzzy controller based on cache hit rates varies the cache size based on the feedback received on cache hits. These two fuzzy controllers are compared against the PID (Proportional, Integral, Derivative) existing algorithm. This controller in \[22\] was proven to be a fairly good per-class, dynamic controller, which makes it great for comparisons.

**Phase II – Squid Proxy Implementation**

For the seconds phase, I implemented the algorithms within the architecture of the Squid web cache proxy (a real world proxy), which is written in the C programming language. The implementation and testing of the Squid web cache proxy was done in two parts. First, modifications were made to Squid’s Windows open source implementation and recompiled. Second, a virtual network was built, while using the Web Polygraph traffic generator to benchmark the implementation.

**Code Modifications**

Squid required three major areas of code modifications. The release of Squid that was modified was a Windows compatible version of Squid 2.7.STABLE6. In order to run the Squid web proxy, a Linux simulated platform on top of Windows called Cygwin was used to compile the Squid source code after the modification. Squid was also run through
Cygwin. The three components were the code related to the file system, the cache replacement policy, and the client-side statistics.

**File System**

Squid defaults to a non-asynchronous storage mechanism called ufs. The following files and functions needed to be modified:

- File: store_dir_ufs.c; Function: storeUfsDirMaintain

  This function was modified with the fuzzification and defuzzification logic where the sizes of cache 1 for class 1 and the size of cache 2 for class 2 are calculated. Both the implementation of the fuzzy controller based response times and the fuzzy controller based on cache hit rates was completed. This function then called the cache replacement policy to remove any extraneous items that would result from making either cache 1 or cache 2 smaller.

**Cache Replacement Policy**

Hewlett-Packard in 1999 had already deviated from the LRU default behavior of Squid by creating a heap implementation of multiple cache removal policies including Least Frequently Used with Dynamic Aging (LFU-DA) and Greedy Dual-Size Frequency (GDSF). I was able to take advantage of their groundwork by implementing my own optimized cache removal policy based on time of use, frequency of use, and page size in their customized heap.

The following files needed to be completely reworked to take into account the extra information, such as web page size and multiple classes for relative differentiation:

- File: store_heap_replacement.c
- File: store_heap_replacement.h
• File: store_repl_heap.c

All the items in the cache (whether class 1 or class 2 related) resided in the same heap. An example of one of the modifications was that the function that removed items from the heap needed to be able to differentiate between the classes for each item. If the item on the top of the heap was class 1 and we wanted to only remove class 2, then items were removed until a class 2 item was found and the removed class 1 items were put back onto the top of the heap after the class 2 item was purged. These modifications made it possible to refrain from modifying too much of the code.

**Client-side**

Since Squid was written without quality of service differentiation in mind, there was no method of outputting the response times and cache hit rates of each class into files. There are many free tools for Squid that output reports, but they only output the overall cache hit rate and response times of the proxy. This was not enough for my statistical needs. The following files and methods were changed to accommodate my needs:

• File: client_side.c; Function: clientUpdatecounters

The client side of the Squid web cache proxy was modified to output each class’s response times and cache hit rates as well as the average weighted response times and cache hit rates into separate files every 100 requests.

**Miscellaneous**

Any other modifications to the source code such as to structs and typedefs were byproducts of the aforementioned changes.
Network Configuration

Due to limited resources, necessity became the motive for creative innovation. Using one Windows machine, I was able to create a virtual network that could sufficiently test the Squid web cache proxy. The following describes the process of configuring the network and testing the system.

**Microsoft Virtual PC**

The goal was to create a small network to test the proxy using one Windows machine. Multiple solutions were explored including VMWare and using shares of cloud computing. Cloud computing can become expensive and VMWare does not run that well when multiple machines are started on the same limited host (only 2 GB RAM).

Therefore, a simpler solution was decided upon. Microsoft provides a free tool called Microsoft Virtual PC that allows the everyday user to start up a Microsoft based virtual machine. Microsoft also separately provide free images of time and space limited Windows XP that web developers use to test different versions of Internet Explorer. I was able to create two virtual machines that were not too taxing on the host when they were both up. Each of these virtual machines acted as places where the traffic generation tool could both requests and serve web pages. Since the traffic generation tool does not require much space or cached memory, these virtual machines were a perfect fit.

**Microsoft Loopback Adaptor**

The Squid proxy resided on the host machine because a Squid proxy can take a lot of resources; the limited virtual machines I had created would not be able to sustain the proxy. However, the problem was that the virtual machines and the host machine all needed to be on the same network without touching the outside world. The solution to
this was to download and install the Microsoft loopback adaptor that allows host machines and virtual machines to easily talk to each other.

**Network**

The test bed network consisted of two virtual machines that acted as traffic generators with the Squid proxy residing on the host machine. The network looked like the following:

![Figure 2 Squid Web Cache Proxy Test Network](Image)

Each machine was configured with the following IP information:

- **IP:** 192.168.200.1, 192.168.200.2, or 192.168.200.3
- **Subnet Mask:** 255.255.255.0
- **Default Gateway:** 192.168.200.254

Each virtual machine ran with one simulated server and one simulated client that either received or generated requests. One virtual machine represented class 1 requests while the other virtual machine represented class 2 requests.

**Web Polygraph Traffic Generator**

I have researched various load generators such as Surge, Curl-loader, Performance Co-pilot, and Web Polygraph. I have also looked into using trace files.
Since Web Polygraph is considered the standard for benchmarking Squid, this became my load generator. Multiple sources agreed that Web Polygraph was quite good at generating realistic web traffic.

Web Polygraph consists of two parts. It has a client with multiple robots and a server. For this experiment, one client and one server were started on each of the virtual machines. Each client was routed through the Squid web cache proxy to its own server. One virtual machine represented class 1 requests while the second virtual machine represented class 2 requests.

The measure of success would be an increased cache hit rate and a decreased response time for class 1 requests when compared to the default Squid configuration.
CHAPTER 5

PERFORMANCE EVALUATION

The key performance metrics providing the ability to compare algorithms are user-perceived response time and cache hit rate. For the simulation, the goal is to demonstrate that a self tuning fuzzy controller when compared to a PID controller can increase the cache hit rate, decrease the response time, and provide a more stable per-class hit differentiation. For the Squid web cache proxy implementation, the goal is to demonstrate that the self tuning fuzzy controller when compared to the Squid web cache proxy default configuration can provide relative QoS differentiation while still improving the overall quality of service, providing a lower a response time, and providing a higher cache hit rate. The self tuning fuzzy controllers, one based on response time feedback and the other based on cache hit rate feedback, were the new creation for this thesis.

The performance evaluations are split into two separate phases that compare both response times and cache hit rates. The phases are:

- a theoretical simulation implementation
- a Squid web cache proxy implementation

The theoretical simulation implementation compared the self tuning fuzzy controller based on response times, the self tuning fuzzy controller based on cache hit rates, and the PID (Proportional, Integral, Derivative) controller. The experiment was split into two sections:

- a simulation using a theoretical traffic generator
a simulation using the 1998 World Cup trace files

Each of these experiments were split into yet another two categories:

- relative QoS differentiation with a requested 2:1 ratio
- relative QoS differentiation with favor to the first class, but no strict ratio

The reason for having a 2:1 ratio versus no ratio is to first prove that close to a 2:1 ratio can be achieved while still keeping an excellent low weighted average response time and a high weighted average cache hit rate. A 2:1 ratio may not be perfectly achieved because the algorithm tries to make sure that the overall performance of the proxy is not hampered by trying to enforce the ratio. The overall performance is defined to be the weighted average response time and the weighted average cache hit rate of the differentiated classes at any point in time. Overall performance is considered good when the average weighted response time is low, while the average weighted cache hit rate is high. If differentiation of classes is not a factor, then the overall performance is the response time and cache hit rate at any point in time in the web cache proxy. Not having the 2:1 relative differentiation proves that the algorithm naturally tends to favoring higher classes while maintaining differentiation, and the weighted average response time and cache hit rate is better than the 2:1 ratio because the self tuning fuzzy controller has more control over tuning itself.

The Squid web cache proxy illustrated only the 2:1 differentiation ratio using a load tool called Web Polygraph. The experiments compared the self tuning fuzzy controller based on the response time, the self tuning fuzzy controller based on the cache hit rate, and the default Squid configuration.

The subsequent graphs and explanations depict:
• the simulation results using a theoretical traffic generator
• the simulation results using the 1998 World Cup trace files
• the Squid cache proxy results using Web Polygraph

**Theoretical Simulation**

The goal of the theoretical simulation included decreasing the response time and increasing the cache hit ratio for the self tuning fuzzy controller based on the response time and for the self tuning fuzzy controller based on the cache hit rate when compared to the PID (Proportional, Integral, Derivative) controller. The fuzzy controllers also worked to achieve a more distinct and stable differentiation between classes.

The theoretical simulation consisted of a Java program that simulated each of the controllers with a theoretical traffic generator that simulated traffic within user-specified parameters. These parameters were composed of the number of websites available, the average arrival time of the request, and the size of each of the websites. Each of these parameters was randomized in order to create a range of workload.

Each of the theoretical runs had the following parameters:

<table>
<thead>
<tr>
<th>Table 1 Simulation Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of websites</td>
</tr>
<tr>
<td>Average website size in MB</td>
</tr>
<tr>
<td>Duration of run in seconds</td>
</tr>
<tr>
<td>Average arrival time interval in milliseconds</td>
</tr>
<tr>
<td>Total cache size in MB (both caches)</td>
</tr>
</tbody>
</table>
Each of these graphs shows the average results of 100 runs. 100 runs were completed of each configuration in order to provide the true average performance. The first set of runs used a relative differentiation constant of 2:1, in which class 1 would try to achieve service that was twice that of class 2. These runs attempt to prove that the fuzzy controllers can achieve a better 2:1 differentiation ratio than PID.

This set of algorithms intrinsically leans toward giving better service to the higher class; therefore, the second set of runs shows the results of running without any ratio. By definition, relative differentiation can be achieved as long as the higher class has equal or better service than the lower class. This experiment attempts to prove that allowing a fuzzy controller to have more control of itself will increase the cache hit rate and decrease the response time, while still achieving relative differentiation.

**2:1 Ratio Relative Differentiation**

The consequent graphs depict the results of the 2:1 ratio for the self tuning fuzzy controller based on response times, the self tuning fuzzy controller based on cache hit rates and the PID (Proportional, Integral, Derivative) controller. The goal is to prove that the self tuning fuzzy controllers are able to achieve the 2:1 ratio with more accuracy than the PID controller. Also, the self tuning fuzzy controllers are to have a higher cache hit rate and a lower response time.

**Fuzzy Controller based on Response Times**

The following graph shows the different in response times between class 1 and class 2. The algorithm was not able to achieve a perfect 2:1 ratio because it was trying to optimize the cache by making sure that the weighted average response time of the cache
did not suffer due to just trying to achieve the 2:1 ratio. The mechanism that kept the weighted average response time low is the algorithm in which a class is given more space if the percentage of requests dropped (requests dropped / total requests) from a class is greater than the other classes. However, the goal was achieved that close to a 2:1 ratio was achieved at a stable rate. Cache 1 did receive notably better service than cache 2.

Figure 3 Fuzzy Controller based on Response Times – 2:1 Theoretical Simulation – Response Times

During the same run, the cache hit rate was also recorded. The following figure shows that the cache hit rate was very close in achieving the 2:1 ratio. The hit rate was even more solid than the response times in terms of stability.
Relative differentiation was fairly easily achieved. This experiment was able to prove that with a fuzzy controller based on response times, pushing a 2:1 ratio in the response times will impact the cache hit rate by creating close to a 2:1 ratio in the cache hit rate as well. The fact that both the cache hit rate and the response time do not have the exact same ratio brings the assumption that response times can affect the cache hit rate, but the two metrics are not perfectly proportional to each other.

**Fuzzy Controller based on the Cache Hit Rate**

The following illustrates that this controller was still able to achieve relative differentiation, but the differentiation between classes was not as distinct as in the fuzzy controller based on the response time in a simulated environment. Class 1 did not have as good a quality of service as the previous controller; however, what service class 1 lost, class 2 was able to gain. These results lead me to believe that controlling the cache hit
rate does not always affect the response time proportionally. An attempt at a 2:1 ratio in the cache hit rate does not always affect the response time in a 2:1 ratio.

![Controller Response Times](image)

**Figure 5** Fuzzy Controller based on Cache Hit Rates – 2:1 Theoretical Simulation - Response Times

The same is true for the cache hit rate where better service was received for class 2, but the differentiation was not as strictly maintained as in the self running fuzzy controller based on the response time. Also, the cache hit rate seemed to react a bit too quickly to changes in the cache due to the oscillations seen in the graph.
The self tuning fuzzy controller based on the cache hit rate was also able to achieve a relative differentiation. However, it did not match the 2:1 ratio as well as the self tuning fuzzy controller based on the response times. Overall, the goal of stable differentiation close to the goal was achieved.

**PID (Proportional, Integral, Derivative) Controller**

Since the PID controller embodies a respected controller in industry, it was used to benchmark the new fuzzy controller based on response times and the fuzzy controller based on cache hit rates against an already existing algorithm. The response times were fairly good, but the 2:1 ratio representing relative differentiation was almost impossible to achieve. Relative differentiation was achieved in the sense that class 1 received better quality of service than class 2, the algorithm was quite difficult to tune to different types of traffic because it was not a fuzzy PID controller. This controller was a basic controller based on the work of [22].
The cache hit rate of the PID controller also exhibited the same behavior as the response time in which differentiation was achieved and excellent service for both classes was achieved, but a 2:1 ratio was not reached due to the difficulty in tuning the controller.
Even though the PID controller was unable to achieve a perfect 2:1 relative ratio between classes, it did have one advantage. The algorithm was able to recognize that both class 1 and class 2 were outputting close to the same load. As a result, it was gravitating towards giving both the classes the exact same ratio, which decreased the weighted average response time and increased the weighted average cache hit rate. By achieving this state, the overall performance of the PID controller would be better than the self tuning fuzzy controllers.

**Controller Comparisons**

Unfortunately, since the PID controller could not achieve the 2:1 ratio, it did have the upper-hand in optimizing the cache space, which gave it an advantage in this simulation. The PID controller achieved marginally better results than the two fuzzy controllers in this case because it did not strictly follow the 2:1 ratio as the two fuzzy controllers did. Both fuzzy controllers achieved close to the same overall response times, while PID did slightly better by not even 1 millisecond.
The cache hit rate played out in the same fashion. PID did slightly better than the fuzzy controller due to the loss of differentiation. The PID controller was closely followed by the fuzzy controller based on the cache hits, which was then followed by the fuzzy controller based on response times.
These overall comparisons bring the conclusion that pursuing strict differentiation can have a negative impact on the overall performance (the weighted average response time and the weighted average cache hit rate). The increase of quality of service for the top classes can hurt the weighted average response time and cache hit rate. In this case, this is due to having the exact same workload for both classes, but expecting a 2:1 ratio. By enforcing the 2:1 ratio, class 2 is forced to have less cache bringing about a lower cache hit rate and a higher response time.

Relative Differentiation

Relative differentiation is defined to be quality of service in which a higher class of service receives equal or better service than a class below it. The following demonstrates the intrinsic ability of the algorithm to provide better service for the higher classes without providing a strict relative service differentiation ratio.
Once again, the self tuning fuzzy controller based on response times, the self-tuning fuzzy controller based on cache hit rates, and the PID controller are compared. The goal is to allow the self tuning fuzzy controller to have more freedom to regulate itself, which should result in a higher weighted average cache hit rate and a lower weighted average response time than the PID controller without losing relative differentiation, in which the higher classes receive better quality of service than the lower classes.

**Fuzzy Controller based on Response Times**

Without the 2:1 ratio, the self tuning fuzzy controller based on the response times was able to balance itself out to produce a better quality of service for class 1 when compared to class 2. On the other hand, the negative impact of not having a ratio defined resulted in greater oscillations of the response time, which was the result of a less stable cache. Since the controller did not have a specified range of service, it reacted more quickly to changes in the response time feedback.
The cache hit rate of class 1 stays slightly better than cache 2 the entire time that the simulation is running. In this simulation, both cache 1 and cache 2 were receiving close to the same number of requests, which caused both classes to gravitate towards each other to optimize the cache hit rate. The control of the response times had the side effect of allowing the cache hit rate to also exhibit the symptom of reacting a bit too quickly to changes in the response time, which caused increased oscillations.
I reached the goal of achieving relative differentiation with a higher overall cache hit rate and a lower response time at the cost of increased instability in the cache. The self-tuning aspect of the controller gave the controller more freedom to pinpoint the load and correctly allocate cache space, but it also gave it the freedom to overreact to small changes in the feedback mechanism.

**Fuzzy Controller based on Cache Hit Rates**

The goal here was the same as the fuzzy controller based on the response times. It is expected that the fuzzy controller will achieve a lower response time and a higher cache hit rate than PID, while still maintaining relative differentiation between classes.

The following two graphs are a prime example on how response time and cache hit rate are not directly proportional to each other. The response times in the first figure did not show much differentiation. The one good thing that was illustrated in the first figure was that class 1 was not overwhelmed by class 2.
Since the cache hit rate was used to keep the differentiation, the cache hit rate showed either at or better service (though slightly) for class 1. With this algorithm, the cache hit rate was kept extremely stable through the entire simulation, and the cache hit rate was completely optimized because the two classes were receiving close to the same workload.

Figure 13 Fuzzy Controller based on Cache Hit Rates - Theoretical Simulation - Response Times
In this case, the algorithm optimized the cache space by providing close to the same service for both classes. Relative differentiation was still reached even though there was no significant differentiation between classes because they both received the same amount of service.

**PID (Proportional, Integral, Derivative) Controller**

In the following two figures, PID was able to keep differentiation between the two classes. The response times were quite good for both classes, and the algorithm was able to recognize that both classes of service had close to the same workload, and it was able to adapt.
However, its cache hit rate oscillated quite a bit more than the fuzzy controller based on cache hit rates. These oscillations exist in this graph even though it was done over 100 runs, which shows how PID may not always be able to adapt to radically changing traffic. This same observation has been made by other researchers as well in the storage industry [22].
Even though the PID controller oscillated quite a bit in terms of response time and cache hit rate, the controller was able to maintain a differentiation of service between classes.

**Controller Comparisons**

In the case where the controller has the ability to manipulate the cache size with only the restriction that a higher class must receive *at or better* service than a lower class, both the fuzzy controller based on response times and the fuzzy controller based on cache hit rates achieve a better response time than the PID controller.
The same case exists for the cache hit rate in which the fuzzy controller based on cache hit rates did the best followed by the fuzzy controller based on response times with the PID controller coming in last. The differences are not extreme, but it is enough to accumulate a better quality of service over an extended period of time.
The goal was reached of proving that in a theoretical environment, the fuzzy controllers could achieve a lower response time and a higher cache hit rate than the classic PID controller.

1998 World Cup Trace File Simulation

Since the 1998 World Cup trace files are used repeatedly to test the efficiency of caches, I decided it would be best to use these trace files to test my own simulated caching algorithm. These trace files do not have a high variation of different websites, but they do present a high workload and can be applicable to the storage field as well. For these runs, one class was given the even IP addresses while the other class was given the odd IP addresses. This may have resulted in creating close to the same workload for both classes.
By utilizing these traces, I was able to surmise that my algorithm thrives in situations where there is less cache space and higher workloads. The space in this cache was very small with only 1 MB of total memory, since the traces are better applied to storage caches. As the amount of cache space lessens or as the congestion gets higher, the algorithm performs increasingly better. Keep in mind that this data only shows the results of one day (day 37) and does not present averaged data; therefore, the oscillations will seem quite a bit more than the results depicted in the theoretical traffic generation simulations.

The goal of these runs is to show that using real-world data, the self tuning fuzzy controllers can achieve a lower response time and a higher cache hit rate than the PID controller without loss of differentiation. The first runs are based on enforcing a 2:1 ratio between classes, while the second set of runs only has the restriction that the first class must receive at or better service than the classes below it.

2:1 Ratio Relative Differentiation

The fuzzy controller simulator and the PID controller simulator were given instructions to try to maintain a 2:1 ratio between class 1 and class 2. The goal was to prove that the fuzzy controllers could achieve a lower response time and a higher cache hit rate than PID without loss of differentiation.

Fuzzy Controller based on Response Times

Even though the response times were used maintain differentiation, it was the cache hit rate that showed the most promise in showing the 2:1 differentiation. Minor differentiation was achieved in terms of response time in that class 1 did receive a
slightly lower response time than class 2. By trying to affect the response time, the cache hit rate was inadvertently drastically affected, even though it seems that the response time was not as drastically affected. This is yet another example showing that the two metrics are not directly proportional. A small change in one metric, such as response time, can have drastic implications on other metrics, such as the cache hit rate.

Figure 19 Fuzzy Controller based on Response Times – 2:1 1998 World Cup Simulation - Response Times
The cache hit rate did achieve very close to a 2:1 ratio. The oscillations occurred as the controller was attempting to react to varying degrees of bursty workload. Even though the results were not as drastic in the response time as hoped, the goal of relative differentiation was achieved even though it was not a perfect 2:1 relative differentiation in terms of response time.

**Fuzzy Controller based on Cache Hit Rates**

As expected, the fuzzy controller based on cache hit rates was able to efficiently allocate all the cache space and to keep the 2:1 ratio requested by the user in relation to the cache hit rate. However, managing the cache hit rate did not seem to have a direct impact on the response times. The response time of the first class stayed par or minorly better than the second class.
The cache hit rate once again exhibited the required 2:1 ratio, even though it did oscillate quite a bit while reacting to the cache hit rate feedback.
Altogether, the goal of relative differentiation was achieved, but the response times did not drastically benefit from the better cache hit rate. This could have been in part due to the fact that there was not a drastic penalty when a miss occurred in the cache.

**PID (Proportional, Integral, Derivative) Controller**

The PID controller had the same problem that the fuzzy controller based on response times and the fuzzy controller based on cache hit rates had. This problem was that the response times seemed unaffected while the cache hit rate clearly showed differentiation. An educated guess would bring one to the assumption that the cache miss penalty was not high enough to result in drastic responses for the response time.

![PID Controller](image)

*Figure 23 PID Controller – 2:1 1998 World Cup Simulation - Response Times*

However, PID did have an extra flaw that the 2:1 was not quite reached for the cache hit ratio to the level that the 2:1 ratio cache hit ratio was in the other two controllers.
This means that if the cache miss penalty became higher in relation to time, then the response times of the PID controller would not be as good as the fuzzy controllers.

Controller Comparisons

Although PID did not achieve the 2:1 ratio as the other two fuzzy controllers did, it did not gain anything in terms of response time or the cache hit rate. The two fuzzy controllers did clearly better in the following figures. The fuzzy controller based on the cache hit rate had a better response time followed by the fuzzy controller based on the response time and finally the PID controller. Even though differentiation was not drastically achieved for any of the algorithms, the higher cache hit rate of the fuzzy controllers resulted in slightly better response times.
The response times were affected by the cache hit rates. Once again the order of controllers from best to worst was the fuzzy controller based on cache hit rates, the fuzzy controller based on response times, and then the PID controller.
In a real-world trace, the 2:1 ratio received better results than in the theoretical simulation. The two fuzzy controllers outperformed the PID controller. The response times were not greatly affected, but a higher cache hit rate can translate into a lower response time when heavy load occurs and the cache miss penalty becomes higher. This observation would lead to even better results in tough workload conditions.

**Relative Differentiation**

In this implementation of relative service differentiation, the algorithms favored the higher classes without specifying a ratio. The goal of this run was to show that relative differentiation could be achieved for the fuzzy controllers in that the higher classes receive at or better service than the lower classes, while still lowering the response and increasing the cache hit rate when compared to the PID controller. The next couple of graphs show that the fuzzy controller based on cache hit rates outperformed the fuzzy controller based on response times followed by the PID controller.

**Fuzzy Controller based on Response Times**

Once again as in the 2:1 ratio experiment for the fuzzy controller based on response times, the response time seemed highly unaffected, but the cache hit rate was drastically affected. The response time of class 1 did however receive slightly better service than class 2, even though it was almost negligible.
Cache 1 was able to achieve a considerably higher cache hit rate than cache 2, while still keeping close to the same response times. In this case, the controller self tuned itself to achieve better than a 2:1 ratio as the ratio started approaching 3:1.
Once again I came to the conclusion that the response times were not highly affected because the cache miss penalty was not high enough to warrant differing response times.

The goal of reaching a higher differentiation ratio for the cache hit rate was achieved even though the response times did not show great differentiation.

**Fuzzy Controller based on Cache Hit Rates**

The fuzzy controller based on cache hit rates was quite disappointing in that the two classes achieved close to the same response times and cache hit rates. No clear differentiation between the two classes was defined, but the fuzzy controller did not break from the rule of relative differentiation because it only had to achieve *at or better* service at higher classes than lower classes.

![Figure 29 Fuzzy Controller based on Cache Hit Rates - 1998 World Cup Simulation - Response Times](image)
The cache hit rate attained almost the exact same rate for both classes since a 2:1 ratio was not enforced. The rules of relative differentiation were appeased because class 2 did not receive better service than class 1.

![Controller Cache Hit Rates](image)

**Figure 30 Fuzzy Controller based on Cache Hit Rates - 1998 World Cup Simulation - Cache Hit Rates**

The results of this run were not as expected. Both classes gravitated towards the same response time and the same cache hit ratio. The 2:1 ratio did not exist to balance out the tendency of the two classes to gravitate to the same response times and cache hit rates since the workloads were close to the same. The positive result was that the rules of relative service differentiation were not broken as both classes received about the same service.

**PID (Proportional, Integral, Derivative) Controller**

PID was clearly able to keep differentiation between classes in the cache hit rate, but the trend continued that the response times seemed highly unaffected. The response
times were able to keep equal service between class 1 and class 2, which kept with laws of relative differentiation.

![Controller Response Times](image)

**Figure 31 PID Controller - 1998 World Cup Simulation - Response Times**

PID was a bit more conservative than the fuzzy controllers in setting its differentiation between classes. While the fuzzy controller based on the cache hit rates achieved close to a 3:1 ratio, the PID controller stayed closer to 2:1 even though the 2:1 ratio was not strictly defined.
The fuzzy controller based on the cache hit rates did not achieve a great differentiation between classes because it was more focused on optimizing the cache. The fuzzy controller based on response times and the PID controller did achieve differentiation even though a ratio was not defined. In the next section, one can see that the fuzzy controller based on the cache hit rate made the right decision for optimizing the cache space available even though a higher range of differentiation as not reached.

**Controller Comparisons**

When the average response times and cache hit rates of each of the controllers was analyzed, the fuzzy controller based on cache hit rates clearly came out on top for the real-world trace. The fuzzy controller based on response times came in next while the PID controller came in last. I think these results may be in part closely related to the fact that the trace files did not contain many sites, but it provided an environment for high
workloads. In times of high workload with a small subset of sites, forcing a higher cache hit rate can have better results than trying to manage the response times directly.

Another observation is that that fuzzy controller based on the cache hit rates did not try to achieve a highly stratified differentiation. When the fuzzy controller was left to tune itself, it gravitated towards providing the same service to both classes, which was advantageous in increasing the overall cache hit rate.
The goal of achieving a lower response time and a higher cache hit rate for the fuzzy controllers over the PID controllers was conquered as the fuzzy controller based on the cache hit rate came in first, the fuzzy controller based on the response time came in second, and the PID controller came in third.

**Squid Web Cache Proxy**

The Squid web cache proxy has been open source for the past decade. This makes it quite difficult to modify due to the lack of documentation and due to the many changes incorporated from various contributors.

In order to test the Squid web cache proxy, Web Polygraph was used running a workload called Polymix-2. This workload has multiple stages including:

- warming up
- slowly increasing to a peak
• decreasing back into a valley
• running idle for a period of time
• increasing once again to a peak
• decreasing once again
• cooling off

The traffic generation executes at about 100 requests per second per client. Each of these clients has 5 robots that generate requests. In our case, class 1 and class 2 clients were both instructed to generate the same amount of load (100 requests per second per client).

For the fuzzy controllers, the Squid web cache proxy was instructed to keep a 2:1 QoS differentiation ratio between class 1 and class 2 requests. The default squid configuration has no differentiation; therefore, no ratio was defined.

The following are some of the results in which the fuzzy controller based on the response time and the fuzzy controller based on the cache hit rate were implemented in comparisons to the default squid configuration.

The goal is to prove that the fuzzy controller based on the response time and the fuzzy controller based on the cache hit rate can achieve a lower response time and a higher cache hit rate for the higher classes while still not taking a dive in performance.

**Fuzzy Controller based on Response Times**

The next graph exhibits the overall trend of the response times for both class 1 and class 2 as the traffic generator begins and continues through one cycle of Polymix-2.
Relative differentiation is achieved (though minor) as class 2 has a slightly higher response time. My assumption is that the 2:1 ratio was not clearly followed because the network used in this experiment was so small that the cache miss penalty in terms of time was almost negligible. As the cache hit rate decreased, the response time slowly increased.

The place where differentiation is clearly seen between classes is the cache hit rate. Once again, a 2:1 ratio was not clearly achieved as in the theoretical examples, but class 1 did receive better service than class 2, which meets the criteria of achieving relative differentiation. The longer the simulation ran, the closer the cache hit rate became to achieving the 2:1 ratio. The differentiation between cache hit rates did not become clearly defined until the overall cache started to stabilize, which was at about 3000 requests. The cache needed to be warmed up with a history of data before it could correctly tune the correct cache hit rate.
The response times did not provide much in the way of clear differentiation, but the cache hit rate did show differentiation the longer that it ran. In an environment where the cache miss penalty is higher, the response times would show a higher form of differentiation, which would follow the results of the cache hit rate.

**Fuzzy Controller based on Cache Hit Rates**

The two subsequent graphs tell close to the same story that the fuzzy controller based on cache response time gives. The response time of class 1 is negligibly better than class 2.
However, in the figure depicting the controller hit rate, one can see that the cache hit rate maintains a clear differentiation between classes the entire time that the Squid proxy is run. Class 1 clearly gets a better cache hit rate. My assumption on the reason that the response time is not clearly affected is because this test network is a small network with almost no penalty when a cache is missed.
The response times did not provide much in showing differentiation, but the cache hit rate performed even better than the fuzzy controller based on response time. The fuzzy controller based on the response time only started showing differentiation close to the end of the run, while the fuzzy controller based on the cache hit rate exhibited clear differentiation during the entire run.

**Default Squid Web Cache Proxy**

The default Squid proxy configuration behaved as expected. Since no differentiation was defined and the traffic generation was the same from class 1 and class 2, then both class 1 and class 2 had the exact same response times. The cache hit rates also maintained close to the same average for both classes. The goal was to use the default Squid configuration as a baseline, since the Squid proxy is already considered fairly optimized in its default configuration.
In terms of the cache hit rate, the Squid web proxy took turns in having a slightly higher cache hit rate. This behavior was expected since the workload was the same for both classes of service.
The Squid web cache proxy provided the expected behavior that all classes would be treated equally using an LRU cache replacement policy. This baseline will be used to compare the weighted average response time and cache hit rate with the fuzzy controllers.

**Controller Comparisons**

The next three graphs portray the comparisons of the weighted average response times, the weighted average cache hit rates, and the cache hit rates for class 1. The response times remained overall (the weighted average between class 1 and class 2) the same for all the controllers. I believe this is largely due to the fact that the test network that conducted the experiment does not introduce high penalties when a cache miss occurs.

![Controller Response Times](image)

**Figure 41 Squid Proxy - Controller Comparisons - Response Times**

Due to the fact that the default squid configuration did not have to achieve a differentiation ratio between class 1 and class 2, it was able to achieve slightly better results. However, the fuzzy controller based on the cache hit rates was able to gain the best of both worlds by not only keeping the differentiation, but also by being comparable
in performance to the default squid configuration. The cache hit rate decreased slowly as time continued because the Web Polygraph generator increasingly requested a wider variety of websites.

![Controller Cache Hit Rates]

Figure 42 Squid Proxy - Controller Comparisons - Cache Hit Rates

One of the goals of altering the Squid web cache proxy was to allow class 1 to have better service than class 2 without hurting the overall cache hit rate of both classes. The following shows the cache hit rate for class 1 between the controllers. The fuzzy controller based on the cache hit rate was able to keep a better hit rate than the default configuration. Surprisingly, the self tuning fuzzy controller based on the response time did poorly. In the previous theoretical simulations, this behavior was not apparent, but there was some indication that it could occur. The theoretical simulations usually had the fuzzy controller based on the cache hit rates doing better than the fuzzy controller based on response times, which was clearly indicated here in a real-world implementation.
The goal of achieving relative differentiation occurred, but it was at the expense of decreasing the performance a bit. The fuzzy controller based on the response times behaved poorly in this situation, but the fuzzy controller based on the cache hit rate was able to achieve the best of both worlds by keeping relative differentiation without having a high hit on the overall performance (average weighted response time and average weighted cache hit rate).

In this case, the goal of achieving better overall performance (low weighted average response time and high weighted average cache hit rate) was not achieved. At least the fuzzy controller based on the cache hit rate was able to achieve close to the same performance as the default Squid configuration.
CHAPTER 6

CONCLUSIONS

One can learn the behavior of algorithms from both simulation and real-world implementation; taken as a whole, the algorithms presented in this thesis were a success. The key metrics for this thesis were response time and cache hit rate. The goal was to create fuzzy controllers that could decrease the response time and could increase the cache hit rate while still maintaining relative service differentiation.

The theoretical simulations did have higher success than the Squid web cache proxy implementation. The theoretical simulation was able to achieve a lower user-perceived response time, a higher cache hit rate, and more controllable differentiation. The Squid web cache proxy had more success in the fuzzy controller based on the cache hit rate in that it could achieve differentiation without negatively impacting the overall cache hit rate and the overall response time when compared to the default Squid web cache proxy configuration.

The theoretical simulation showed that the fuzzy controller based on the cache hit rate and the fuzzy controller based on the response time outperformed the PID controller even though they carried a higher differentiation between classes. The fuzzy controller based on the cache hit rate consistently showed more promise than the fuzzy controller based on the response time, which indicated that it might also do better in the Squid web cache proxy implementation.

PID was extraordinarily difficult to tune, and it was a lot more difficult to get PID to reach a 2:1 differentiation between classes. Tuning the target hit ratio for each class
became a game of trial and error to optimize the cache for a certain strain of traffic. This was due to this controller not being a fuzzy self-tuning PID controller, but a classic PID controller based upon the model set forth in [22].

The authors of [22] also made use of the 1998 World Cup trace files in their simulation to prove that their algorithms were better than PID in the storage industry. Since the 1998 World Cup files are used in both the web cache proxy world and the storage world, this implies that many of the algorithms for both these worlds are shared.

Since my algorithms become more efficient with higher workloads and lower cache sizes, this makes this set of algorithms a candidate for moving over to the storage arena.

The Squid web cache proxy implementation of the fuzzy controller based on the cache hit rate was able to keep par with the default Squid configuration. The fuzzy controller based on the cache hit rate performed the best, while still maintaining a relative differentiation between classes.

In general, the goal of this thesis was achieved of lowering the response times and increasing the cache hit rate when compared to other established algorithms such as the PID (Proportional, Integral, Derivative) controller and LRU (Least Recently Used).
REFERENCES


Information Technology, Dubai, May 2006, pp. 1-5.


