Scheduling on Web Servers

By

Yanxiao Chen

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of the
Master of Computer Science

Ottawa-Carleton Institute of Computer Science
(OCICS)

Carleton University
Ottawa, Ontario, Canada K1S 5B6
December 2002

©Copyright 2002, Yanxiao Chen

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79762-7

Canada

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of the thesis

Scheduling on Web Servers

Submitted by Yanxiao Chen
in partial fulfillment of the requirements for
the degree of the Master of Computer Science

Dr. Shikharesh Majumdar, Thesis Supervisor

Dr. Frank Dehne, Director of the School of Computer Science

Carleton University
December 2002

Abstract

New technologies such as secure sockets layer (SSL), dynamic requests and middleware applications, combined with cheaper bandwidth and modems, increase the demands on the Internet. In addition, an increasing numbers of images, audios, videos and cgi applications are embedded into web pages. As a result, popular web server systems are frequently heavily loaded; requests are either rejected, or the user-perceived latency is dramatically increased. Thus, there is a need for effective approaches for improving the performance of web serves.

This thesis explores the possibility of achieving high performance of web servers through scheduling polices that prioritize requests in accordance with the number of embedded objects in a web page. The purpose is also to provide insights into the various factors that affect the performances of the scheduling policies.

The experimental environment is a three-tier system that consists of clients, a dispatcher and multiple replicated back-end servers. The dispatcher and back-end servers are the popular Apache web servers that process requests with different priorities. Through the experimental results, we conclude that giving higher priorities to the web pages with the lowest number of embedded objects can achieve substantial performance benefits over a neutral first-come-first-served policy. This performance benefit can be furthered improved by giving a lower priority to the request for the main HTML file than the request for an embedded object within the same web page. Therefore, by using such web page characteristic-based scheduling polices, the overall performance of the system can be improved.

Acknowledgments

I would like to express my sincere thanks to my supervisor, Professor Shikharesh Majumdar, for his valuable guidance, encouragements and efforts that put into the thesis. In addition, I am also grateful for the learning experience I am provided by my supervisor.

I would like to express my gratitude to the National Science and Engineering Research Council of Canada (NSERC) for providing the financial supports for this research.

I would also like to extend my truthful appreciation to Diwakar Krishnamurthy for his help in understanding the performance measurement tool, to Rosemary Daniels for her reading the draft version of this thesis, to my parents and people in my church for their great encouragements.

TABLE OF CONTENTS

Lists of Figures.....	vii
List of Tables.....	viii
Glossary of Terms	ix
Chapter 1 Introduction.....	1
1.1 Background	1
1.2 Motivation	2
1.3 Goals of the Thesis	3
1.4 Thesis Contributions	4
1.5 Thesis Organizations	6
Chapter 2 Literature Review	8
2.1 Web Workload Characterization	8
2.2 Request Parallelization to Achieve High Performance Web Servers	10
2.3 Parallel Job Scheduling	11
2.4 Priority-based Techniques to Improve Web Server Performance.....	12
2.4.1 Admission Control	13
2.4.2 Scheduling Policies	19
2.4.3 Backgrounding Mechanism.....	25
2.4.4 Web Content Adaptation.....	27
2.4.5 Resource Containers.....	28
Chapter 3 Platform and Tool for Prototype.....	29
3.1 Apache Web Server.....	29
3.1.1 Process Management.....	30
3.1.2 Persistent Connection.....	32
3.1.3 Modules.....	33
3.2 Scheduling Policies in Linux OS	35
3.2.1 Real-time Process Scheduling.....	36
3.2.2 Conventional Process Scheduling	37
3.2.3 Pitfalls of the Linux Scheduler.....	39
3.3 Httpperf -- A Tool for Web Server Performance Measurement	40
Chapter 4 Performance Prototype	45
4.1 System Architecture	45
4.1.1 Clients.....	46
4.1.2 Dispatcher and Back-end Servers	47
4.2 Workload and System Parameters.....	54
4.3 Server Configuration Parameters	56
4.4 Scheduling Policies	58
4.5 Performance Metrics	60
4.5.1 Mean Response Time	60
4.5.2 Mean Slowdown.....	61

4.5.3	Differentiation Factor	62
4.6	Validation of the Prototype	62
4.6.1	Applying Little's Law	62
4.6.2	Arrival Rate vs. Mean Response Time.....	64
4.6.3	Performances of High vs. Low Priority Web Pages.....	65
4.6.4	User-specified and Measured Arrival Rates.....	67
4.7	Implementation Issues.....	67
Chapter 5	Results of Experiments	69
5.1	Impact of Workload Characteristics.....	69
5.1.1	Impact of the Arrival Rate.....	69
5.1.2	Impact of Service Demand.....	75
5.1.3	Impact of the Number of Embedded Objects.....	78
5.1.4	Impact of Web Page Class Popularity.....	83
5.2	Impact of the Number of Back-end Servers.....	86
5.3	Priority for Web Page Components	89
Chapter 6	Conclusions	94
6.1	Summary	94
6.2	Conclusions	95
6.3	Future Research.....	97
References		100
Appendix A		104
Appendix B		105
Appendix C		106
Appendix D		107
Appendix E		108

Lists of Figures

Figure 1-1: Experimental Environment.....	4
Figure 2-1: Cookie-based AC Architecture (from [38])	18
Figure 2-2: Application Level Queuing Model (from [16]).....	22
Figure 3-1: Multi-process-based Server Model	30
Figure 3-2: Sample of a Httpperf Session	42
Figure 4-1: System Architecture	45
Figure 4-2: Sample Code to Set Process Priority.....	49
Figure 4-3: Sample Code to Generate the Service Demand.....	51
Figure 4-4: Performance of Requests for a Web Page Class with a Fewer Numbers of Embedded Objects.....	66
Figure 4-5: Performance of Requests for a Web Page Class with a Larger Number of Embedded Objects.....	66
Figure 5-1: Impact of Arrival Rate for Fixed Service Demand	70
Figure 5-2: Impact of Arrival Rate for Exponential Service Demand	70
Figure 5-3: Impact of Arrival Rate on Mean Slowdown	74
Figure 5-4: Impact of Service Demand on the Performance of LNEF	75
Figure 5-5: Impact of Service Demand on the Performances of Scheduling Policies	76
Figure 5-6: Performance Comparison of Scheduling Policies for Different Service Demands.....	77
Figure 5-7: Impact of the Number of Embedded Objects on the Performance of LNEF	79
Figure 5-8: Impact of the Number of Embedded Objects on the Performances of Scheduling Policies	80
Figure 5-9: Performance Comparison of Scheduling Policies for Different Number of Embedded Objects.....	81
Figure 5-10: Impact of Web Page Class Popularity on the Performance of LNEF	84
Figure 5-11: Impact of Web Page Class Popularity on the Performances of Scheduling Policies	85
Figure 5-12: Performance Comparison of Scheduling Policies for Different Web Page Class Popularities	86
Figure 5-13: Impact of the Number of Back-end Servers on the Performance of LNEF	87
Figure 5-14: Impact of the Number of Back-end Servers on the Performances of Scheduling Policies	88
Figure 5-15: Performance Comparison of Scheduling Policies for Different Number of Back-end Servers	89
Figure 5-16: Impact of Using Priority for Web Page Components.....	90
Figure 5-17: Performance of LNEF_LPH (a) $E_1=2$ (b) $E_1=3$	93

List of Tables

Table 4-1: Number of Iterations to Generate Service Demands	52
Table 4-2: Workload and System Parameters	55
Table 4-3: Apache Server Configuration Parameters	56
Table 4-4: Little's Law Verification	63
Table 4-5: Mean Response Time at a Very Low Arrival Rate.....	64
Table 5-1: The Comparison of Fixed vs. Exponential Service Demand.....	72
Table 5-2: Impact of the Arrival Rate on Differentiation Factor for Fixed Service Demand	73
Table 5-3: Impact of the Arrival Rate on Differentiation Factor for Exponential Service Demand	73
Table 5-4: Performance of Requests for Web Page Classes Achieved with FCFS and LNEF	82

Glossary of Terms

AC	Admission Control
CGI	Common Gateway Interface
DF	Differentiation Factor
FCFS	First Come First Served
FIFO	First In First Out
HNEF	Highest Number of Embedded Objects First
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
LAN	Local Area Network
LNEF	Lowest Number of Embedded Objects First
LNEF_HPH	LNEF policy that gives a <i>higher</i> priority to a request for the main <i>HTML</i> file in comparison to a request for an embedded object
LNEF_LPH	LNEF policy that gives a <i>lower</i> priority to a request for the main <i>HTML</i> file in comparison to a request for an embedded object
QoS	Quality of Service
RAM	Random Access Memory
SRPT	Shortest Remaining Processing Time
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
URL	Uniform Resource Locator

Chapter 1 Introduction

1.1 Background

Over the past few years, the Internet has undergone a phenomenal growth without a sign of stopping. New technologies such as SSL, dynamic requests, data transactions and middleware applications dramatically increase the service demands of the Internet. Commercial companies try to distinguish their sites from others by the heavy use of multimedia. Consequently, the Internet has increasingly become an important facet in people's lives for email, web surfing, online shopping and banking.

The traditional non-priority-based web server and the "bursty" workload characterization dramatically limit a server's performance when it is heavily loaded. The poor performance can be seen both from the server's and from the client's perspective [10]. From the server's perspective, the response time increases until its capacity is reached, and the throughput of the server is then flattened. From the client's perspective, a long queuing delay at a server can lead him/her to give up and abort the request. More and more companies rely on the Internet to conduct and manage their global business, and long delays over the Internet mean losing profits and customers. Also, more and more individuals and organizations are willing to pay higher Internet access fees to benefit from applications such as Internet telephony, videoconferencing and online games. This creates a need for a high performance web system.

System performance has two components: network performance and server performance. A high network performance ensures that certain demands for bandwidth and network delay are met. A high server performance guarantees that a reasonable client response time is achieved. A complete Internet solution can never be achieved without the supports from the server system since a highly loaded server can become a bottleneck and reject the requests from clients.

1.2 Motivation

Priority-based scheduling policy at a web server is one of the many research areas explored, in order to achieve high performance web servers. However, no attempts have been made to prioritize requests according to web page characteristics, such as the number of embedded objects, which is addressed in this thesis.

A typical web page consists of a main HTML document and multiple embedded objects, such as images, text files, frames, as well as embedded audios, videos and cgi applications. New technologies undertake continuous improvements in modem speeds and available bandwidths. With high-speed modems and increased bandwidth, Internet clients tend to increase the size of web documents and to download large files. For instance, an average of 17 embedded objects per web page is observed in a study of popular web sites [32]. The requests for files with a large number of embedded objects may push the performance bottleneck to the server system and can lead to an increase of client perceived latency for a web page.

Parallelizing web page access by storing the embedded objects on separate web servers that can be accessed concurrently has been shown to improve system performance significantly [32]. There is a further need for using a desirable scheduling policy in such a parallel web server environment that is capable of handling large traffic volumes and improving the overall performance. This thesis focuses on this scheduling problem.

1.3 Goals of the Thesis

One purpose of the thesis is to investigate the possibility of achieving high performance through desirable scheduling policies, in which requests are prioritized according to the number of embedded objects in a web page. This is achieved by establishing a parallel server system that consists of clients, a dispatcher and multiple replicated back-end servers. A request for the main HTML file or an embedded object in a web page can thus be served by any one of these servers. Multiple embedded objects in a web page can be accessed in parallel with the help of such a replicated server set. Although data replication used in this research is for improving performance, a replicated set of servers can improve system reliability as well. The studied scheduling policies are supported in the middle layer (the dispatcher) and in the last layer (the back-end servers). The other purpose of the thesis is to study the effects of workload characteristics on the performances of the different scheduling policies.

1.4 Thesis Contributions

This research builds a prototype and investigates its performance under various workload parameters and scheduling policies. The experimental environment used for this purpose is presented in Figure 1-1. The system consists of a 100MB Ethernet LAN that interconnects a number of Pentium II PCs that incorporate the clients, the servers and the dispatcher. A more detailed description is provided in Section 4.1 of Chapter 4.

Figure 1-1: Experimental Environment

This prototyping and measurement approach allows us to capture the system overheads that are difficult to capture accurately in a simulation or an analytic model-

based environment. The important contributions of this research are briefly summarized:

- ♦ The thesis describes a design and an implementation of a prototype that consists of re-engineered Apache web servers and synthetic clients running on a Linux operation system. The scheduling policies supported by the web servers include:
 - First Come First Served (FCFS)
 - Lowest Number of Embedded Objects First (LNEF)
 - Highest Number of Embedded Objects First (HNEF)
 - LNEF policy that gives a *Lower* Priority to a request for the main *HTML* file in comparison to a request for an embedded object of a web page (LNEF_LPH)
 - LNEF policy that gives a *Higher* Priority to a request for the main *HTML* file in comparison to a request for an embedded object of a web page (LNEF_HPH)
- ♦ Significant performance benefits can accrue from giving a higher priority to the web page with the lowest number of embedded objects (the LNEF policy) for certain types of workload.
- ♦ Using different priorities for web page components can substantially affect the overall performance. The performance benefits of the LNEF policy, which gives equal priority level to the HTML document and the embedded objects within the same web page, tends to gradually decay with the increase of the number of the

embedded objects in a web page. This can be overcome by using a lower priority for the main HTML file and a higher priority for the embedded objects within the same page (the LNEF_LPH policy).

- ♦ The performance of HNEF is the most sensitive to the change of the workload characteristics, such as the service demand of a request, the number of embedded objects in a web page as well as the popularity of a web page class. HNEF is also highly reactive to a change of resources such as the number of back-end servers in a system. LNEF demonstrates the slowest reaction to change and FCFS is placed in the middle. The effect of workload characteristics and system resources is most substantial at a high system load.

The results of this research are applicable primarily to a system in which web servers host various web pages that differ in the numbers of embedded objects.

1.5 Thesis Organizations

Chapter 2 is a review of the techniques to achieve high performance web servers that are reported in the literature. Workload characterization on typical web servers is first introduced, and approaches to achieve high performance web servers are then presented. These approaches include parallelization of the web page retrieval, embedded object replication and load balancing among back-end servers. Issues associated with parallel job scheduling are then addressed. The web server performance improvement techniques, such as admission control, scheduling policies,

backgrounding mechanism, web content adaptation and resource containers, are described in greater detail.

The platform and the tool for the prototype are presented in Chapter 3. This includes the Apache web server, the scheduling policies supported by the Linux OS, and httpperf, a workload generation and performance evaluation tool. The reengineering of httpperf to meet the specific needs of this thesis is also addressed.

The performance prototype is presented in Chapter 4. The system architecture, workload and system parameters, system configuration, and scheduling policies are described, followed by the performance metrics. A description of system validation is also included in this chapter. Implementation issues are addressed as well.

Chapter 5 presents the experimental results. The experiments consider the effects of the changes in workload parameters and the use of additional system resources (the number of back-end servers).

Chapter 6 presents the conclusion of the thesis and suggests directions for future research.

Chapter 2 Literature Review

This chapter surveys techniques to achieve high web server performance. Workload characterization and its implication on web performance are first introduced in Section 2.1. Section 2.2 discusses the techniques for achieving high performance web servers, such as parallel access of multiple embedded objects in a web page from multiple back-end servers, balanced distribution of embedded objects and replication of embedded objects on back-end servers. Section 2.3 addresses issues associated with parallel job scheduling. Priority-based web server performance improvement techniques are further discussed in section 2.4. These techniques consist of admission control, scheduling policies, backgrounding mechanism, web content adaptation and resource containers.

2.1 Web Workload Characterization

Research [5, 6, 7] has focused on understanding the web workload and its impacts on performance. Data are usually gathered by analyzing proxy or server logs of popular sites. Key observations are summarized as follows:

- ♦ Over 90% of the requests are for static files (HTML and image files).
- ♦ Most files transferred are small (the median is less than 5KB), but a few large files (audio, video, compressed and executables) have significant impacts on

web traffic by bytes. The mean transfer size is 21KB. This large variation of file size is known to give rise to heavy tailed distribution.

- ♦ More and more large files are downloaded by users as a result of the increase in available bandwidth.
- ♦ Web traffic is self-similar, which means that bursts exist on all time scales. Although self-similarity does not appear in all workloads, it does appear in heavy traffic. Crovella [15] explains that the causes of self-similarity might arise from the heavy-tailed distribution of transmission time (primarily due to the distribution of web file size), the influence of user think time between page requests, as well as the caching effects (larger files tend not to be found in the cache).
- ♦ The set of files that users are interested in change over time, but some files retain their popularity for extended periods.
- ♦ Rejecting a request means the closing of the connection. Closing a connection is expensive and the resources it consumes are almost 70% of the resources which otherwise are used to serve a small file [37].

Research on web server performance enhancement has provided a number of insights that are briefly described:

- ♦ File caching (on browser or on proxy) can improve performance substantially. Caching a small number of most frequently referenced files (they are usually small in size) improves cache hits. However, a tradeoff exists between caching large number of small files to increase hit rate and caching a few large files to reduce the number of bytes transferred [5].

- ♦ Merely adding up the available bandwidth does not relieve the workload crunches. Users' download behavior gradually places a heavier load on web servers, and proxy caching is increasingly important under such circumstance [7].
- ♦ Proper cache replacement policies and prefetching techniques to detect the change of popular documents are promising for performance enhancement.

2.2 Request Parallelization to Achieve High Performance Web Servers

Nadimpalli [32] explores the possibility of parallelizing web page requests in order to improve web server performance. The techniques she describes include retrieving embedded objects in a web page from multiple back-end servers in parallel, server replication, and load balancing policies used by the dispatcher.

The server system used in Nadimpalli's experiment consists of a dispatcher and multiple back-end servers. The main observations presented in [32] are summarized:

- ♦ Substantial performance improvement is observed by accessing multiple embedded objects in a web page concurrently from multiple back-end servers.
- ♦ Balanced distribution of embedded objects within a web page on back-end servers is crucial to harvest the benefits of retrieval parallelization. Unequal distribution degrades system performance substantially.
- ♦ Replication of embedded objects on back-end servers has a similar performance improvement as balanced distribution.

- ♦ Different service demands of each embedded object within a web page (intra-page variability) affect the performance. Higher intra-page variability is observed to downgrade the benefits of parallelism in web page access.
- ♦ Inter-page variability due to differentiated service demand of different web pages also affects the system performance. Higher performance benefits accrue from parallel-embedded-object-retrieval when larger inter-page variability exists in the system.
- ♦ Different policies for dispatching requests to back-end servers affect system performance. A Round Robin strategy can produce higher performance than a Random strategy.

2.3 Parallel Job Scheduling

A significant amount of knowledge exists in the area of scheduling parallel jobs (typically scientific applications) on multi-programmed parallel systems. A survey of such scheduling techniques is provided in [29]. A number of these scheduling strategies use job characteristics in determining the priority of a job. This thesis however focuses on a web environment in which the dynamics of the service of a web page request is different from the execution of jobs on a multi-programmed parallel system. A parallel scientific program, after being submitted, is run until it completes. On a web environment, however, the request for the main HTML file in a web page is sent first; the subsequent requests for the embedded objects are generated

after the response to the first request reaches the client browser. New research on scheduling on a web environment is warranted.

2.4 Priority-based Techniques to Improve Web Server Performance

Priority-based techniques to improve overall web server performance are discussed. In general, the need for such techniques comes from the following situations:

- ◆ **Prioritizing Requests**

Companies in E-commerce might give higher priorities to requests that tend to finalize a transaction [28] (such as credit card payments) than those requests that are just surfing the web.

- ◆ **Prioritizing clients**

Subscribers to a web site might be favored over those browsing for free.

- ◆ **Web hosting service**

A web server system powerful enough to manage multiple sites for different organizations might want to guarantee services (such as outgoing bandwidth) to clients in proportion to the money each one pays [2].

Priority-based techniques consist of a classification of requests into groups of various service requirements, request admission control, group-based resource allocation and schedule prioritization.

Request classification (such as classifying requests to premium users and regular users) is the first step before any priority-based techniques can be provided. There are two types of classification [10]:

- ◆ Client-based classification

This is based on client attributes such as client IP address, cookies embedded in the requests, and special client identifications embedded in requests by browser plug-ins. It can be used to differentiate preferred clients from others.

- ◆ Target-based classification

This is based on attributes such as URL and server IP addresses (used when multiple virtual sites are co-hosted by the same server). This classification can be used to provide better service to some requests, such as e-commerce purchase or virtual sites that pay premiums for better service.

Once classification is complete, admission controls and other priority-based schemes can be used. These schemes will be discussed in following subsections.

2.4.1 Admission Control

Effective admission control (AC) mechanisms should assure that high priority requests are not dropped before low priority requests. Response time is also a concern. *Tail-dropping* (requests are dropped when number of waiting requests exceed some threshold value) is not suited for a highly variable workload environment in which traffic bursts occur in all time scales [12]. Research by Bhatti and Friedrich [10] focuses on more flexible AC schemes. They base their AC on two threshold values. Basic requests are rejected under two circumstances: if the total number of queued requests is above a threshold, and if the number of premium

requests exceeds another threshold. More complicated ACs are discussed in subsequent sections.

2.4.1.1 Estimation-based AC

Chen et al. [12] introduce their periodic AC based on estimating request rate and service time. It bounds response delay in dynamic environments. In their strategy, estimates of periodic request rate of each priority group are based on a historical access pattern. Service time is estimated based on the document type (used to estimate the size) of the request. The estimations help to allocate resources to various priority groups.

The admission criteria demands that total service time of all accepted requests does not exceed the predicated system capacity, and that the total service time of the priority group (to whom the request belongs to) is below the capacity allocated to that priority group. With more resources allocated to a high priority group, the performance of this group is ensured by restricting admission of lower priority requests.

It is possible that high priority requests are dropped due to lack of queue space while lower priority requests are starving in the queue waiting for the services. A double queue structure overcomes this weakness [12]. A request is first sent to an AC manager, which classifies the priority and decides whether it is enqueued based on the criteria. Enqueued requests wait in the primary queue. At the beginning of next

period, unfinished requests are moved to a secondary queue. When the secondary queue is full, it is cleared up and requests in this queue are dropped.

The request scheduler picks up the requests and sends them to the server pool. No requests in the secondary queue can be picked up until the primary queue is empty. In this structure, since the primary queue is quickly served, a new request (in the primary queue) can be served without waiting for a long time, and hence bounded delay is achieved for most of the requests (most requests are small requests). This technique allows the flexibility of dynamic change of resource allocation and fast movement of the pending requests.

2.4.1.2 Measurement-based AC

Li and Jamin [26] present an algorithm to allocate a pre-configured fixed percentage of bandwidth to numerous simultaneous clients, independent of the aggression of some clients. The aggressive clients fill up the socket queue and squeeze out preferred clients. Their algorithm does not directly focus on dropping requests, but on deferring non-preferred clients so that CPU can be assigned quickly to the next request from the queue.

The AC configuration is specified at server startup. The total bandwidth (the maximum amount of bandwidth available to all clients) and percentage allocation to each client class are specified as the configured parameters. These parameters are read into the shared memory, and a Bandwidth Manager is responsible for their

maintenance. The bandwidth information is measured and updated each time a request comes.

A delay value is used to indicate whether a request should be processed immediately or not. A request is delayed if a client requests more resources than it has been allocated and when the server is fully utilized. If the server is idle or partially saturated, more bandwidth can be granted to clients who need more. In the saturated case, the aggressive client's available bandwidth will be forced down to the allocated bandwidth, and more bandwidth is allocated to those who need it.

If the number of concurrent delayed processes exceeds some limit when a new request arrives, the child process handling the request is immediately killed and this request is rejected. In this way, other clients can be served more effectively.

2.4.1.3 Session-based AC

Session-based AC is triggered by the growth of E-commerce in which a sequence of activities such as browsing the catalogues, selecting products, purchasing and making payments, is a session of individual requests to web servers. During the overloading, dropping a session in the middle of some requests not only wastes server resources, but more importantly, it adversely affects a company's profitability. Cherkasova [14] argues that an overloaded server discriminates against a longer session which makes the situation worse since a longer session means a higher possibility of the client making a purchase.

The session-based AC presented by Cherkasova aims at giving a fair chance of completion for any accepted session, independent of the session length. Thus, only

when a server has the capacity to process all future requests related to the session, will it accept a new session.

The AC in [14] is based on server CPU utilization. The CPU utilization is measured within a time interval, such as each second. An “observed” utilization is then computed based on both this measured value and server load prehistory, which is controlled by a coefficient k (between 0 and 1). If $k = 1$, the “observed” value is based entirely on the measured value in the last time interval; if $k = 0$, the “observed” value is based entirely on the server load prehistory; the smaller the value of k , the smaller the impact of the last measured value. The observed utilization is then compared to a threshold value (e.g. 95%). If it is above the threshold value, all new sessions are rejected and the server only serves the admitted sessions. After the observed utilization drops below the threshold, the server allows new admissions in the next time interval.

2.4.1.4 Cookie-based AC

Voigt et.al [38] propose a session based AC which does not reject all clients when the server load exceeds a threshold. It aborts connections that are considered less important. They introduce a kernel based AC in which service differentiation is based on filter rules associated with connection and information in the http header, such as cookies, URL or the types of requests.

Sessions that are considered important or sessions that are initiated by premium users are allowed to complete the sessions even during an overloaded condition. Session information can be found in the URL or cookies. When a HTTP

reply is sent back to a client, a cookie indicating the level of importance of the session is attached. In the next request of the same session, the cookie is automatically inserted into the HTTP header. Whenever the session changes its level of importance, the updated cookie is sent to the client.

Three levels of service differentiation are identified in the network stack of the operating system. The first level bases information entirely on TCP and IP headers (i.e. the source and destination). The second level obtains application level information such as URL or type of the request (e.g., CGI request). For instance, a lesser admission is given to CGI requests in order to reduce the load. The third level makes use of information on cookies. Filter rules with string comparison are used on each level. Filter rules are installed into the kernel.

The architecture used is shown in figure 2-1.

Figure 2-1: Cookie-based AC Architecture (from [38])

The khttpd handles static content and any dynamic content is passed to the Apache web server. The khttpd parses the http header and passes the URL and cookie

information to the QoS module, which makes use of its filter rules and returns the admission decision to khttpd. The first two levels of filter rules can be used to decide the admission of initial requests (for instance, whether CGI requests should be admitted); the cookie-based rules are used to decide whether to abort or continue a session. The QoS module regularly measures the CPU utilization, and cookie-based overload control is used only when the CPU utilization is higher than a threshold. In such a context, only persistent connections that carry cookies specifying that the connection is important are allowed to proceed.

2.4.2 Scheduling Policies

Prioritized scheduling of system resources can be either accomplished on application-level or on kernel-level. A number of scheduling techniques available in the literature are discussed in the following subsections.

2.4.2.1 Work-conserving and Non-work-conserving

Almedia et.al.[2] study application-level and kernel-level priority-based scheduling to achieve a differentiated service for web content hosting clients. Preemptive scheduling at kernel level and non-preemptive scheduling at application level are considered.

For the application-level scheduling, an Apache web server is modified so that an additional child process is spawned to act as a scheduler. The priority is classified

according to the customer name embedded in the URL. For the kernel-level approach, both Apache and Linux source codes are modified.

The scheduling policy considers two components: the sleep policy and the wakeup policy. The sleep policy decides whether to process a request immediately or to postpone the processing. The wakeup policy decides whether a postponed request should continue. The scheduler maintains a queue of postponed requests. The two policies are implemented by using thresholds for the maximum number of requests that can be concurrently handled in each priority level. Hence, there are fixed number of slots for each priority level and each coming request either occupies a slot (to execute) or waits in a queue (is blocked), which depends on whether a non-work-conserving policy or a work-conserving policy is used.

The non-work-conserving policy allows a request to proceed if the total number of requests in the same priority slot is below the threshold and no requests are allowed to occupy slots of different priority levels. The work-conserving policy allows requests to occupy slots of different priority levels.

Almedia et. al. conclude that restricting the number of concurrent processes is a simple and an effective strategy in obtaining differentiated services. A non-work-conserving policy works better for multiple levels of priority. In a work-conserving policy, an overflow of low priority processes into high priority slots might occur, which would result in the loss of performance differentiation. A careful design of the application level approach is good enough, although a kernel-level approach is always better for heavily loaded servers.

2.4.2.2 - Size-based Scheduling

For “static requests, the size of a request (i.e. the time required to serve the request) is well-approximated by the size of the file, which is well-known to the server” [23]. This section introduces the studies undertaken in order to prioritize the requests in accordance with the file sizes of the requests.

2.4.2.2.1 Shortest-Remaining-Processing-Time Scheduling

When a task size is known, the Shortest-Remaining-Processing-Time (SRPT) scheduling is widely understood as a strategy to minimize mean response time. However, SRPT scheduling is considered to favor short requests and starve long requests. Nevertheless, Harchol-Barter [23] demonstrates that under heavy-tailed web workloads, even the largest requests are not penalized by SRPT scheduling. Suppose we want to compare heavy-tailed and exponential distributions. Based on the heavy-tailed characteristics, the largest 1% of file requests accounts for over 50% of the load and they are interrupted by less than 50% of the total load. However, for the exponential distribution of the same mean, the largest 1% of file requests count for only 5% of the total load and they are interrupted by 95% of the total load. Therefore, under heavy-tailed distribution, the large file requests suffer less than other distributions, such as exponential distribution.

The kernel level and application level implementation of SRPT scheduling have been proposed by Harchol-Barter [23] and Crovella [16], respectively.

At the kernel level, Harchol-Barter implements this scheduling policy by modifying a Linux kernel and an Apache web server. The orders in which the socket buffers are drained are controlled. The kernel is modified so that priorities are given to those sockets corresponding to connections for small file requests or where the remaining data required is the smallest.

The study indicates that SRPT scheduling yields substantial performance improvements over fair scheduling (e.g. time-shared) in mean response time, mean slow down (actual response time over the service demand), and variance in response time for most of the requests (99.5% of the requests are small) [23]. Under a traditional kernel's one-socket-queue architecture, a short request could have a really long waiting time in this queue. However, with this new approach, short requests have been separated into much shorter priority queues, which substantially improves the waiting time.

Crovella et.al. [16], on the other hand, implement SRPT scheduling at the application level (see Figure 2-2).

Figure 2-2: Application Level Queuing Model (from [16])

Once a connection is accepted through an *accept* system call, the single listen thread is blocked. Then it passes the connection descriptor to the protocol queue and continues to listen to new connections. The protocol thread parses the request and unfolds the file size. It enqueues the connection descriptor to the disk queue. The disk thread dequeues the connection descriptor and *read* file data into the connection descriptor's buffer. Once this is done, the descriptor is enqueued into the network queue. The network thread dequeues the descriptor and *writes* the data onto the associated socket's buffer. Once all the bytes have been transferred, the connection is closed; otherwise, the descriptor is put back into the disk queue.

The connections are prioritized in the disk and network queues according to the least number of bytes remaining to be served. The protocol queue is served in FIFO order since the sizes of the files are not known yet. The network is observed as the bottleneck in Crovella's experiments. Crovella shows that the mean response time can be improved by factors of four in comparison with Apache-like size-independent policy.

The drawback of the application level approach is its lack of control over the order of requests inside the OS. For instance, the orders in which socket buffers are drained might not be the same as the order in which the application fills these in. This issue is addressed in Lazy Receiver Processing [17]. Traditional interrupt-driven Unix/Linux operating system gives priority from high to low in the order of packet capturing, protocol processing and application level processing. Under a high system load, receiver livelock will occur [24, 30]. It means that when highest priority is given to interrupt handling, the system spends all of its time responding to interrupts, and

those packets are only discarded later because no CPU time is assigned to process on the application level. Lazy Receiver Processing is a network architecture that ensures that incoming requests are scheduled at the priority of the process that will handle the request. It allows the scheduling to be made on a per-connection basis at the network interface level.

2.4.2.2.2 Alpha Scheduling

Cherkasova [13] proposes another size-based priority policy (without preemption) which is based on a coefficient alpha. It resides in the middle of FIFO and shortest-job-first. The alpha value helps to re-order the priorities of the requests. Unlike shortest-job-first, this scheduling is starvation free in the sense that no request is delayed indefinitely by other requests.

In the alpha scheduling, the requests belonging to the same class (they have the same priority) are served in the order of their arrivals. However, the requests belonging to different classes are served according to the calculated priorities. If a request r arrives, its priority is calculated as $p(r) = c + \alpha * \text{size}(r)$, where $\text{size}(r)$ is the file size retrieved by request r ; c is a parameter (a running-queue clock) that starts at zero and is incremented by $\text{size}(x)$ if a request x is served; α is any value between 0 to infinity. c is reset to zero if all requests are served. A small priority value $p(r)$ implies a high priority. $\alpha=0$ means that it is a FIFO strategy; a finite positive value of α means each request is eventually served. Larger α gives better response time since it enlarges the priority gap between small request and large request.

The author shows that the short-request-first strategy improves overall response time per http request over a FIFO strategy by more than three times in a heavily loaded situation.

2.4.2.3 Lottery Scheduling

Lottery scheduling [39] supports proportional allocation of resources, and it can be used for providing differentiated services for each client. It manages a system's resources with the help of tickets. Each client is assigned a number of lottery tickets proportional to the expected share of resources allocated to it. Whoever holds the winning ticket is granted the resource. A winning ticket can be randomly generated. Thus, lottery scheduling probabilistically allocates resources to competing clients in proportion to the number of tickets that they hold.

The major drawback of lottery scheduling is that it does not map the application-level performance targets, such as the expected mean response time, to the allocation of system resources necessary to achieve the targets [4].

2.4.3 Backgrounding Mechanism

Eggert and Heidemann [18] present a server-side application level mechanism that relies on the concept of background and foreground requests. Background requests are low priority, preemptable requests whose existence do not affect the performance of foreground requests. An example of background requests could be embedded image files of a web page. The main HTML document which is given

higher priority can be served as a foreground request, while those embedded image files are treated as background requests. Another example could be prefetching, which is undertaken periodically in background.

The main idea of backgrounding is to slow down the background requests and make more resources available to foreground requests. In their implementation, different sockets for background and foreground requests are used so that the requests are demultiplexed into two different queues. Three mechanisms studied are briefly described:

- ♦ The first mechanism limits the number of background processes. It restricts resource usage by background processes by reducing concurrency.
- ♦ The second mechanism is the first mechanism plus lowering priorities of background processes.
- ♦ The third mechanism allows background processes to explicitly adjust sending rate by pausing. Multiple background processes share a fraction of the rate limit.

The authors of [18] observe that all of the approaches work well, and the third approach is exceptionally better in reducing the variance of response time. The third approach lowers the foreground performance slowly as background loads grow substantially.

2.4.4 Web Content Adaptation

Instead of rejecting incoming requests in the overloaded condition, Abdelzaher et al. [1] suggest the degradation of the content of the page without visibility by clients. The potential resource saving arises from three areas: the quality differentiation of some significantly compressed images is insignificant on most clients' displays; embedded objects in a page can be reduced by using text for some decorating objects (icons, bullets, bars and backgrounds), and restraining the depth of link a client is allowed to browse reduces the load on the server.

A special adaptation tag (e.g., degradable or important) can annotate parts of the contents which are used by a content management tool to create separate HTML versions of the site. Multiple versions of the content for a given URL are supported. For instance, under directory “/root”, two sub-content trees “full-content” and “degraded-content” are maintained. For a URL “/pict.img” can be either served with the content “/root/full-content/pict.img” or “/root/degraded-content/pict.img”. This can be applied to dynamic content as well. Large file space is needed to serve various versions of some popular pages. If the server is underutilized, higher quality images can also be placed on some pages.

A middleware between the server and network system is proposed [8]. This layer intercepts each request and modifies the URL to direct the request to the right content tree according to the load condition.

2.4.5 Resource Containers

The operating system usually undertakes scheduling and allocates resources such as CPU time and memory based on processes or threads as a whole. Processes or threads are accountable entities (called resource principals) for resource consumptions. The promise for such a design is that processes or threads are entities that carry out independent activities. However, in some situations, multiple processes / threads may cooperate to carry out a single independent activity. In other situations, a single process / thread may perform multiple independent activities.

Banga et al. [9] propose the resource container abstraction that decouples the resource principal, such as process or threads, from the resources allocated and provides support for more reasonable resource management and service differentiation in the OS.

A resource container is an operating system abstraction that logically contains all system resources consumed to undertake an independent activity. One container might have multiple threads / processes associated with it. The system scheduler allocates resources in accordance with the resource containers rather than processes and threads.

Chapter 3 Platform and Tool for Prototype

This chapter studies the platforms and the tools used for the prototype: Apache web server, scheduling policies on Linux operating system and httpperf, a workload generating and performance measurement tool. The system setup is a cluster of Apache web servers running under Linux OS. Httpperf is a free software used as the client side. Understanding each of them is important to understand the core of the thesis.

3.1 Apache Web Server

Apache [3] is one of the most widely used web servers today. It can use either multi-processes or multi-threads, depending on the versions and the platforms. Apache 2.0 supports a hybrid of multi-processing and multi-threading on Unix/Linux. Because the release date of version 2.0 occurred after we started to build the system in the thesis, version 1.3.19 is used in this thesis. This version supports multi-processing on Unix / Linux platforms, and a brief introduction for this version is presented in this section.

Figure 3-1 shows the Apache's multi-process-based server model.

Figure 3-1: Multi-process-based Server Model

Important issues associated with the Apache server, such as process management, persistent connection as well as major modules used in this thesis are discussed in the following subsections.

3.1.1 Process Management

A parent process is responsible for launching children processes. It uses a technique called “preforking”. When the parent process is launched, it preforks a certain number of children processes. Each child process listens on a well-known port (default port 80) to serve requests. Each of them makes the system call *accept* on the same listening descriptor.

The parent process creates the listening socket before any child process is spawned, and therefore the file structure referred by this listening descriptor is

duplicated in each child process. After *fork* is called, these children refer to the same file structure. Each child calls *accept* and is blocked by the kernel if no request exists in the listen queue. All sleeping children wake up when the first client connection is established. The first child picks up the request from the listen queue and the other children go back to sleep. Requests in the listen queue are served in the First-In-First-Out (FIFO) order. Further details on the Apache web servers are described by Stevens [34].

Preforking saves the costs of doing one fork per client. However, the number of children to start should be perceived by the parent beforehand. Too few children degrades performance and too many children has the side effect of waking up too many processes each time a connection is to be accepted. Once per second, the parent process performs the maintenance of monitoring the status of all children processes. It forks or kills additional children according to the *MinSpareServers* and *MaxSpareServers* values.

Keep-alive statuses of children are maintained in a shared memory. From the configuration file, the user-defined values of *MinSpareServers*, *MaxSpareServers* and *StartServers* settings affect the process creation. After the initial spawning of *StartServers* children, the parent process creates additional children in the following way: it creates one, waits a second and then creates two, waits a second, then spawns four and continues until it reaches the maximum spawn rate of 32 per second. If more than 8 children are spawned per second, a warning message is written into the log file that suggests that the user should tune these settings. The creation stops until it reaches the *MinSpareServers* setting. The maximum number of idle children

processes is regulated by *MaxSpareServers*. Idle children processes above the limit are killed.

MaxRequestsPerChild controls the number of requests a child can serve before it is terminated by the parent. By default, this value is set to zero, which means that a child can handle infinite number of requests. It is suggested that this value should be set to 10000 in case of memory leak on SunOS or an old version of Solaris [22]. *MaxClients* setting controls how many requests the Apache server can simultaneously handle. The limit is 256 requests.

3.1.2 Persistent Connection

Apache 1.3.19 merely supports HTTP/1.0 if configured as a proxy. It allows a client to send a single request on the TCP connection, and this connection is closed once the response is returned to the client. A typical web page request is composed of multiple single requests, one for the main HTML file and multiple requests for the embedded objects. A browser first sends the request for the HTML file and only after the reply is received, the browser sends the requests for all embedded objects simultaneously. For a page with several embedded objects, HTTP/1.0 increases the page retrieval latency with the additional overheads of opening and closing connections for each single request.

On the other hand, HTTP/1.1 allows multiple requests to be sent on a single connection [21]. The connection is kept open for an interval of time (15 seconds by default) in anticipation of further requests. This persistent connection allows opening

and closing fewer TCP connections, and resource usages (CPU time and memory for TCP protocol control blocks) are thus reduced. Pipelining of requests and responses over a connection allows a connection to be used more efficiently [21]. Sending multiple responses on a single connection increases network utilization with the avoidance of TCP slow-starts [4]. Slow-start is a congestion avoidance technique which allows the TCP congestion window to exponentially increase after each ACK is received until the capacity of the internet is reached [36].

3.1.3 Modules

Separated modules chosen by a user are compiled into an Apache server. In this thesis, in addition to default standard modules, proxy module (*mod_proxy*) and rewrite module (*mod_rewrite*) are explicitly compiled into the dispatcher. The employment and application of these two modules are introduced in the following subsections.

3.1.3.1 Proxy Module

A proxy server sits between the client and the server to forward requests or to provide caching services. It filters sites and can be used as a sort of firewall since it hides the real IP address of the servers. For an absolute URL <http://www.site.com/fool.html>, the proxy either forwards the request to another proxy or requests the relative URL /fool.html from www.site.com. The address of the proxy

is configured into the browser. Apache not only supports this proxy setting, but also supports the reverse proxy setting.

Reverse proxy provides greater flexibility by acting as the actual server www.site.com. It translates the relative URL /fool.html to an absolute URL address of one of its back-end servers, for example, http://www.backendserver2/fool.com. The URL translation is accomplished by the rewrite module (please refer to next subsection). In this way, it not only provides a single access point, but also has a complete control over the back-end server delegation, which is an excellent load balancing technique.

Apache allows reverse proxy to adjust the *Location* header when a response returns from a back-end server. It prevents the client from by-passing the reverse proxy to directly access the servers. Further details of the reverse proxy can be found in [20]

3.1.3.2 Rewrite Module

The *mod_rewrite* module implements a rewrite engine based on regular expression to manipulate URLs. The module is hooked into the Apache server during the compile time and is called for every incoming URL at run time. The module then manipulates the URL for further processing. Its usages such as URL layout, content handling, access restriction are described by the author of this module, Engelschall [19].

In this thesis, the rewrite module rewrites a URL requested by a client into a URL that includes the identity of a back-end server. This is necessary because a page is replicated and is available on all of the servers. The proxy then forwards the request to this back-end server identified by the rewritten URL.

3.2 Scheduling Policies in Linux OS

Linux scheduler offers three scheduling policies, *SCHED_OTHER* for normal processes and *SCHED_FIFO* (First-In-First-Out) and *SCHED_RR* (Round Robin) for real time applications.

The default *SCHED_OTHER* is the time-sharing policy. The CPU time is divided into slices, one for each ready process. If a running process is not terminated when its quantum expires, a context switch takes place and another process in the ready queue is scheduled.

There are two types of priorities: static priority and dynamic priority. Static priority ranges over the values of 0-99 with 99 as the highest priority level. By default, all conventional processes have a static priority of 0. Static priorities of 1 to 99 are reserved for real-time processes. A conventional process is not scheduled to run when a ready real-time process exists in the system. Dynamic priority is often regulated by the “nice value” which ranges from -20 (highest) to 19 (lowest). The kernel scheduler keeps track of the status of each process and adjusts its dynamic priority periodically. A process’s dynamic priority is boosted if it has been waiting for the use of the CPU for a long time [11].

Linux kernel is not preemptive, but processes are preemptive. A process is preempted when any of the following situations occur: the process's time quantum expires, a new process is ready to run and its priority is higher than the currently running process, or a process with higher dynamic priority returns from I/O and will continue to finish its time quantum. The preempted process still stays in the ready-to-run queue.

Section 3.2.1 and Section 3.2.2 further discuss real-time process scheduling and conventional process scheduling.

3.2.1 Real-time Process Scheduling

Only a privileged process is allowed to make the system call *sched_setscheduler* to change the scheduling from the default to real-time policy. The OS maintains a list of ready-to-run processes at each priority level. A ready-to-run process with higher static priority can always preempt the running process with lower static priority. *SCHED_FIFO* and *SCHED_RR* determine who obtains the CPU for processes with equal static priority.

The OS keeps a *SCHED_FIFO* process at the head of the queue of its priority level. If no other real-time process with a higher priority is runnable, the process will continue to use the CPU until it voluntarily yields the CPU or is blocked by I/O. If it is preempted, it will resume execution only after the higher priority running process is blocked or is terminated.

A *SCHED_RR* process is allowed to run for a maximum time quantum (150ms). Upon the expiration of its time quantum, it is put back to the end of queue of its priority level. If preempted before its quantum finishes, it subsequently resumes to finish the remaining portion of its quantum. The time quantum in *SCHED_RR* is not alterable.

Unfortunately, *SCHED_RR* is not demonstrating its expected behavior for kernel 2.2.16 used in this thesis. A running *SCHED_RR* process is not always preempted when its time quantum is exhausted. It favors the currently running process over other processes with the same priority. The problem is discovered and addressed in [27] for the same or higher version of the Linux kernel.

3.2.2 Conventional Process Scheduling

Upon being spawned, a process by default adopts the time-sharing policy with a static priority value of zero. A quantum is assigned to the process and the scheduler relies on hardware timer interrupts. With lower static priority value than a real-time process, a conventional process can be preempted by any real-time process. The dynamic priority decides which conventional process runs next. The initial dynamic priority is 20 ticks. A counter is used to maintain this priority and it is reduced by one tick in every 10ms.

The CPU time is divided into epochs (a period of time). In a single epoch, each process is assigned a time quantum (maximum CPU time that the process can use in this epoch) that is computed at the beginning of the epoch. Each process can

have a different quantum. If a process waits for I/O in the middle of the execution, it can obtain the CPU many times in one epoch as long as its time quantum is not used up. An epoch ends when all runnable processes in the ready queue exhaust their time quanta. Processes that are blocked might still have quantum remaining to be used in the next epoch. Each runnable process is assigned a new quantum when a new epoch starts.

The calculation of time quantum of a process in an epoch is as follows:

- ♦ Each process is initially assigned a base time quantum of 20 clock ticks;
- ♦ A process using up its current quantum gets this base quantum again in the next epoch;
- ♦ If a process does not exhaust its time quantum, the unused quantum is carried over to the next epoch. In the new epoch, the dynamic priority is the base quantum plus the unused quantum. Linux favors I/O bound processes since these processes probably will not finish their quanta.
- ♦ If a child process is forked, the parent's total remaining quantum is cut into half: half for itself and half for the child.
- ♦ A child process inherits the base time quantum from the parent process.

Within its time quantum, a process with higher dynamic priority gets back to the CPU right away after finishing the I/O. In addition, a process waiting for the usage of CPU for a while has its dynamic priority boosted each time this priority is recalculated. The basic time quantum would be reset by the system call *nice* or *setpriority*. The user level's "nice value" within the range of -20 to 19 is converted

into the scale of 1 to 40 (40 is the highest priority and 1 is the lowest priority) at the kernel level. Unlike the conventional process, a real-time round robin process only gets the basic quantum.

3.2.3 Pitfalls of the Linux Scheduler

Pitfalls of the Linux scheduler are addressed in [11]. A short summary is presented in this subsection.

- The algorithm does not scale well when there are a large number of runnable processes:

When the number of processes grows, the overheads of recomputing dynamic priorities tend to be large. Linux reduces the overhead by making the recalculation occur only at the end of an epoch. In this way, the priorities of I/O bound processes (such as the interactive applications) tend to be boosted less often giving rise to long response times. The CPU bound processes in general are still less favored in comparison to those I/O bound processes since they usually do not have remaining quanta to be carried forward to the new epoch.

- ♦ Default quantum not responsive enough for high loads:

The basic time quantum of 210ms for conventional processes is large when the number of runnable processes is high in the system. Less responsiveness occurs in this situation.

- ♦ Priority boosting strategy not optimal:

Some I/O bound processes such as the ones performing database searches are not interactive, but they are unnecessarily boosted. On the other hand, CPU-bounded interactive applications usually suffer.

- ♦ Non-preemptable kernel not well suited for real-time processes:

A runnable real-time process might have to wait for several milliseconds for interrupt handling [11].

3.3 Httpperf -- A Tool for Web Server Performance Measurement

Various tools can be used to conduct benchmark tests on web server performance. The outstanding characteristic of httpperf lies not only in its capability to sustain high offered loads when the server is overloaded, but also in its session-based workload generation and measurement. Both HTTP/1.0 and HTTP/1.1 requests are supported. The persistent connection and pipelining in HTTP/1.1 allow users to send requests concurrently without waiting for their responses.

To accommodate its extensive modification over time, the tool is divided into three separate parts: the core HTTP engine, the workload generator and the statistics collector. The engine takes care of all details regarding communications with the server, which include connection, request generation and reply handling. The workload generator is in charge of initializing HTTP calls at proper times. Finally, the statistics collector is responsible for calculating performance metrics. The different parts interact through event signaling [31].

Httpperf uses the timeout mechanism to sustain overload. A user can indicate a timeout value (in seconds) for a maximum waiting time. This value is used when it establishes a TCP connection, sends a request, waits for a reply and receives the response. If any of this activity fails to make progress within the given timeout value, httpperf closes the connection and increases the client *timo* error count. The default value is infinity. When waiting for a reply, the actual timeout value used is the sum of this timeout value plus a think-timeout value. The think-timeout indicates the maximum time a server needs to start sending the reply. It is very useful for requests from long-running cgi scripts. In this way, httpperf times out requests that have waited for a long duration, which not only releases the load on server side, but also reduces resource usage on the client side (such as the number of TCP connections).

Two kinds of workload generators are supported: a request generator and a URL generator. The request generator allows a user to generate either pipelined calls on a connection or session requests. The URL generator generates a sequence of URLs at a given rate.

A session consists of several bursts, which are separated by user think-times. Each burst includes a certain number of calls. Figure 3-2 demonstrates a sample session.

Figure 3-2: Sample of a Httpperf Session

This session consists of two call bursts. It simulates a user and the browser behavior. A call burst can be seen as a web page request. After the reply for the main HTML document is received, httpperf will send requests for the embedded objects concurrently. The user then pauses for a duration (think-time) before sending another web page request, which is another burst in the same session.

Sessions can be generated sequentially or at some specified arrival rate. For a workload consisting of sequential sessions, multiple machines must be used in parallel to simulate concurrent users. The use of multiple machines to achieve a high workload can be problematic with regard to the availability of the number of machines. On the other hand, if each session is treated as one web page request, specifying a session arrival rate allows us to easily simulate requests from concurrent users. This is the method used in this thesis.

The major drawback of httpperf is that since it consumes all available CPU cycles, only one httpperf process can be run on one CPU. Unnecessary background jobs should also be kept to a minimum [31]. In addition, httpperf is not capable of generating real life traffic directly taken from the log file of some servers.

To modify the operations of `httperf` to meet our specific needs in this thesis, its source codes are revised in the following ways:

- ♦ Transforming `httperf` into an open queuing model

An open model [25] is characterized by the continuous flows of jobs “into” and “out of” the system and it is the most generally used model today.

`Httpperf` allows users to pre-specify the number of requests entering the system according to some specific arrival rate. But the drawback is that once all requests are sent to the servers, no new-coming jobs will enter the system, which is not the open model widely agreed upon.

With the following two steps, `httperf` is modified to support an open model: 1) Categorizing requests into tagged and untagged; 2) Stopping `httperf`.

Tagged requests control the running or the stopping of `httperf`. The system needs to warm up for a while before collection of performance data can begin. The tagged requests are composed of those initially sent out in the warm-up stage, and followed afterward by those whose performances are measured. Untagged requests are allowed in the system to keep up the arrivals of requests into the system until the last tagged request completes. The untagged requests continuously enter the system until `httperf` receives all the replies that it intends to measure.

For instance, if we are interested in measuring the performance of 14000 requests, we can set, for example 14100 requests, as tagged requests; among these, the first 100 requests belong to the warm-up stage. The untagged requests are the requests follow the first 14100 requests until `httperf` stops.

Differentiated request IDs are used to tag the requests. Each web page request is given different ID. If there are N web page requests in a workload, these web page requests will have different IDs from 1 to N . More importantly, the main HTML document contains the same ID as the embedded objects for a web page request.

Httpperf does not terminate until all the tagged jobs are successfully completed. Statistics are then gathered and calculated, and performance metrics are computed.

- ♦ Decomposing the mean response time into smaller pieces

Suppose the workload consists of N web page requests, N_1 are for web page 1 and $N - N_1$ are for web page 2. Httpperf only returns the mean response time of all N requests in general. The reengineered httpperf measures the response time of request for web page 1 and web page 2 separately and also reports a mean response time for a web page request. The decomposition of the general mean response time into small pieces helps to observe the separate performance of different classes in a workload that consists of web page classes characterized by different number of embedded objects.

Chapter 4 Performance Prototype

A detailed description of the performance prototype built is presented in this chapter. Section 4.1 presents the system architecture, which is followed by a description of the workload and system parameters, the server configuration parameters as well as the scheduling policies in Section 4.2, Section 4.3 and Section 4.4 respectively. The performance metrics are presented in Section 4.5 and the results of system validation are included in Section 4.6. Implementation issues are addressed in Section 4.7.

4.1 System Architecture

The performance prototype in this thesis builds upon the Apache web server, one of most widely used web servers today. Figure 4-1 illustrates our three-tier system architecture.

Figure 4-1: System Architecture

Each component of the system (clients, dispatcher and back-end servers) runs on a network of 266MHz Pentium II PCs with 64MB of RAMs under Red Hat Linux 6.2, kernel 2.2.16-3. The PCs are inter-connected by a 100 MB Ethernet LAN. The PCs and LAN form the “quiet network” in the Real Time and Distributed System Lab where experiments run without interference from other users.

Client requests are simulated by a `httperf` process that requests web pages at a specified arrival rate.

The dispatcher is configured as an Apache reverse proxy. The dispatcher and the back-end servers are multiple processes that serve requests from a central queue. The setup of the multiple back-end servers is organized according to the research in [32] that demonstrates that parallelization of embedded objects across multiple back-end servers can enhance system performance.

Each component of the prototype system is described in the following subsections.

4.1.1 Clients

The client machine runs a control program that corresponds to the executable of `httperf`. `httperf` reads web page requests from an input file which is generated beforehand by a C program. Appendix A shows an example of the input file that consists of sessions of requests. Each session contains one call burst to represent the main HTML file and the embedded objects of a web page. The arrival rate is session-

based. For example, an arrival rate of 1.9 req/sec means that the time between the arrival of the successive sessions is 0.53 ($1/1.9$) seconds. The requests for the embedded objects are generated after the request for the main HTML document is answered.

We have used a synthetic workload that consists of two web page classes. Each class is characterized by a distinct number of embedded objects. Which particular web page is requested by httpperf is controlled by the probability that a client will request a specified web page class. Consider a system with two web page classes characterized by 1 and 10 embedded objects. Assume for example that the probability of requesting the web page class with 1 embedded object is 95% (i.e. the web page class with 10 embedded objects has a 5% chance of being accessed). A random number is generated and compared to 0.95. If the number ranges between 0 and 0.95, the client requests a web page with 1 embedded object; otherwise, it requests a web page with 10 embedded objects. Each run consists of 14000 web page requests for all experiments. Our workload construction allows us to measure the impact of the workload characteristics (such as the probability of particular web page class requisitions, or the number of embedded objects in a web page class) on the performances of scheduling policies.

4.1.2 Dispatcher and Back-end Servers

Issues related to the dispatcher and the back-end servers, such as the Apache server architecture, the use of priority-based scheduling policies, the dispatching

policy, the generation of service demands and the overheads from the system are discussed in this section.

4.1.2.1 The Apache Server Architecture

The implementation used on the dispatcher and the back-end servers is aimed at keeping the source code of the operating system untouched and revising Apache as little as possible.

In a multi-process Apache system, server processes handle requests that are placed in a central listen queue. The order of the requests in the listen queue is not alterable at the user level. Requests are given priority only after they are extracted from the listen queue. The dispatcher and the back-end servers maintain the original architecture supported by Apache, as shown in Figure 3-1. The parent process is responsible for spawning children processes and maintaining their status. Children processes are ready to handle requests once spawned. When a child process receives a request from the central queue, it looks up its priority and invokes the system call *sched_setscheduler* for OS to re-schedule. The system call sets the static priority of the child process and the scheduling control is returned to the OS scheduler. After this system call, the child process becomes a real-time process in Linux and is re-scheduled according to the priority value that is set as one of the parameters. Figure 4-2 presents the sample source code that achieves such a real-time scheduling for a child process.

In Figure 4-2, the policy value of *SCHED_FIFO* and the priority value of 99 indicate that the current process will be turned into a real-time process with the

priority-of 99 and all processes with the same priority level will be served in the first-in-first-out order. The scheduler of the OS will then be invoked to make the re-scheduling.

```
int policy=SCHED_FIFO;
struct sched_param param;
param.sched_priority=99;
int mypid=(int)getpid();

if (sched_setscheduler(mypid, policy, &param)== -1)
{
    perror("error in setting priority");
    exit(1);
}
```

Figure 4-2: Sample Code to Set Process Priority

By keeping the architecture of Apache server untouched, the revision to Apache source code is reduced to a minimum.

4.1.2.2 Priority-based Scheduling on Dispatcher and Back-end Servers

Prioritizing requests are meaningful only when requests are queued up on the server. It could occur either on the dispatcher or on back-end servers. Therefore, both the dispatcher and the back-end servers should implement the priority-based scheduling.

Requests for web page components (main HTML file and embedded objects) arrive at the dispatcher from the client. Then requests are forwarded by the dispatcher to the back-end servers. A queue of waiting requests is maintained on the dispatcher and each back-end server. Multiple Apache processes at the dispatcher (or a back-end

server) remove requests from this queue. A process changes its priority in accordance with the priority associated with the web page class it is serving.

The dispatcher and the back-end servers maintain the consistent priority value for each request. Once the dispatcher knows the priority value of the current request, it inserts this value into the header field when the request is forwarded to a back-end server.

4.1.2.3 Dispatching Policy

The Apache reverse proxy can flexibly delegate requests to back-end servers in order to achieve load balancing. In our case, back-end servers are fully replicated, which means that the same data are stored in each server, and any of the servers is eligible for serving any request. In addition to the scheduling policies used for setting the priority of processing a web page component, a dispatching policy that determines which back-end server the request is sent to is required at the dispatcher. The dispatching policy that is used to select a back-end server is Round Robin. For the Round Robin dispatching policy, if there are N back-end servers ranging from server1 to server N , the 1st request is directed to server1, the 2nd request is directed to server2, ..., the N^{th} request is directed to server N and the $N+1^{\text{th}}$ request then goes back to server1 and so on.

This setup benefits from the research in [32] which demonstrates that fully replicated back-end servers achieve a higher performance than back-end servers with an unbalanced load. According to Nadimpalli [32], Round Robin gains better load

balancing than a Random dispatching policy, since, with a Random policy, multiple requests could be directed to the same server while other servers are idle.

As in [32], caching is turned off on the dispatcher and on the back-end servers to sustain the largest workload and obtain better control of the experiment. All requests are HTTP/1.0 requests.

4.1.2.4 Generation of Service Demands

A *for* loop with a system call *getrusage* is used on back-end servers to simulate the service demand associated with the processing of the main HTML file or an embedded object. The execution of the system call burns CPU cycles. The number of iterations used by the *for* loop represents how many times *getrusage* should be called to reach the expected the service demand (e.g. 100ms) of the current request. Figure 4-3 presents the sample code to achieve the service demand of 100ms. Repeating the system call 50000 times to reach the service demand of 100ms arises from the data presented in Table 4-1.

```
int loops=50000;
struct rusage usage;
int i=0;

for(i=0; i<loops; i++)
    getrusage (RUSAGE_SELF, &usage);
```

Figure 4-3: Sample Code to Generate the Service Demand

- **Table 4-1: Number of Iterations to Generate Service Demands**

Number of Iterations	Service Demand (ms)
24,500	49.7
24,600	50.04
24,700	50.2
...	...
49,900	99.3
50,000	100.4
50,100	101.6
...	...
73,900	149.9
74,000	150.3
74,100	150.6

Table 4-1 is generated by measuring the execution time of the *for* loop and the number of iterations is increased in steps of 100. For a given required service time (input), the right column of the table is searched until a number higher than or equal to input is found. The corresponding entry in the left column is used for determining the number of iterations in the *for* loop. The input service demands are reported in the context of the performance results presented in the thesis. Note that the actual service demand generated is slightly higher than the corresponding input value. For an input of 50ms, the number of iterations is 24,600 and the actual service demand is 50.04ms; for an input of 100ms, the number of iterations and the actual service demands are 50,000 and 100.4ms respectively; for an input of 150ms, the number of iterations and the actual service demands are 74,000 and 150.3 ms respectively.

4.1.2.5 - Overheads

Overheads resulting from the implementation or from the server system could, in some degree, affect the overall performance. These are presented in this subsection.

- ♦ Overheads from setting request priority

The system call *sched_setscheduler* is used to invoke the Linux scheduler to re-schedule according to the priorities of all requests once a new request enters the system. Super-user permission is required by this system call. Due to the current policy of the Real Time and Distributed System Lab, root permission is given out only on the machine running the dispatcher. However, on back-end servers, not treated as super users, we are not able to set the priority directly. Hence, we use the combination of system calls *fork* and *execl* on back-end servers.

If a child server process A tries to update its priority to 5, it *forks* a child process A1 which *executes* a program to change process A's priority. After increasing its parent's priority, A1 exists. Process A executes as a user process with the new priority. This program is written in C and installed by the lab administrator with root access.

Each child process calls *fork* and *execl* to update its priority upon serving a request. It calls them again to restore the priority back to the original level once the request has been served. Therefore, each single request consists of two such call combinations. Each *fork* and *execl* combination takes 4.3 ms, and two such combinations take 8.6 ms for each individual request for the main HTML file or for

an embedded object. The minimum request service demand is 50ms in the experiment, which is large in comparison to the overhead. Moreover, since this overhead is present in case of all scheduling policies, it is unlikely to affect the relative performance of the scheduling policies this thesis focuses on.

- ♦ Overheads from HTTP/1.0 requests

All requests in our system are HTTP/1.0 requests. HTTP/1.0 uses a separate connection for each embedded object. It involves closing and opening multiple connections for each web page request; this can become expensive especially for a large number of embedded objects in a web page.

4.2 Workload and System Parameters

The synthetic workload used in the research allows us to effectively control the workload parameters and to observe their effects on performances. The main workload and system parameters are briefly described.

- ♦ λ - is the arrival rate of a web page request (req/sec). The inter-arrival time is exponentially distributed.
- ♦ S - is the mean service demand in milliseconds (ms). It is associated with each main HTML file or an embedded object within a web page. It can be either fixed or exponentially distributed, as indicated by D .
- ♦ D - is the distribution of the service demand (fixed or exponential). For the sake of simplicity, we have used an exponential distribution for introducing variability in service demands. A similar approach was used in [32]. Investigation of

scheduling for systems with a higher variability in service demand can be interesting and needs further investigation.

- ♦ E - is the number of embedded objects in a web page class.
- ♦ WP_i - is the web page class i ($i=1,2$). Each web page class is characterized by the number of embedded objects, and the distribution and the mean of the service demand. Thus $WP_i(E, D, S)$ represents web page class i that contains E embedded objects and has a mean service demand of S milliseconds with a distribution given by D .
- ♦ $Pr[WP_i]$ - is the probability with which a client makes a request for web page class WP_i . This parameter simulates the popularity of a web page class.
- ♦ B - is number of back-end servers. The default value is 4. B is a system parameter.

The values of these parameters used in the experiments are presented in Table 4-2:

Table 4-2: Workload and System Parameters

Parameters	Values
λ	Depends on experiment
S	50ms, 100ms, 150ms
D	Fixed, exponential
E	1, 2, 3, 10
WP_i	WP_1, WP_2
$Pr[WP_i]$	0.9, 0.95, 0.99
B	2, 3, 4. The default value is 4.

The impacts of the given parameters on performances associated with different scheduling policies are investigated in Chapter 5.

4.3 Server Configuration Parameters

For simplicity, the dispatcher and back-end servers maintain the same configuration parameters. Table 4-3 enumerates the values of the configuration parameters, which include the default values and the actual values used.

Table 4-3: Apache Server Configuration Parameters

Parameters	Default Values	Values Used
StartServers	5	20
MinSpareServers	5	20
MaxSpareServers	10	25
MaxRequestPerChild	unlimited	10000
MaxClients	150	1000
Timeout	300 seconds	900 seconds

Most values are tuned higher than the default values provided by the Apache server in order to accommodate the high system load in the experiments. The rationale for choosing these values is briefly discussed.

♦ StartsServers, MinSpareServers and MaxSpareServers

StartsServers represents the number of children processes spawned at the beginning. MinSpareServers and MaxSpareServers are minimal and maximal numbers of idle

children processes at any point of time. While each Apache child process handles requests, the parent process ensures the adequacy of the number of children during a maintenance cycle (each second).

When these values are too low, the request queuing delay and maintenance overhead is increased. When these values are too high, the server resources are not used efficiently. To ensure all experiments can be organized to use the same setting, these values are set high enough to meet our needs. The process of setting the values is based on the study in [32].

- ♦ **MaxRequestPerChild**

This is the total number of requests a child process can handle before it terminates. The setting of this value is suggested by [22] to sustain higher performance without memory leak.

- ♦ **MaxClients**

This is the value that restricts the number of requests a server can simultaneously serve. Clients can be blocked out if the value is too low. This value should be set high [22]. To ensure that the testing can be conducted in a high system load, the limit is raised from 256 to 1000.

- ♦ **Timeout**

This parameter sets a limit on the number of seconds before the server sends a timeout response to a client. A low timeout value will quickly timeout requests. That

this value is set higher than the default value is to ensure that the testing can be conducted for a high system load.

4.4 Scheduling Policies

The dispatcher and the back-end servers use the following scheduling policies (SP):

- ♦ First Come First Served (FCFS)

This is the non-priority based policy. It makes use of Linux kernel's *SCHED_FIFO* real-time policy. Equal static priority of 99 is assigned to all processes that handle requests. All requests are processed in first-in-first-out order under the same priority setting.

- ♦ Lowest Number of Embedded objects First (LNEF)

It favors the web pages with the fewer number of embedded objects. Requests for these web pages are given higher static priority, and thus are processed earlier than those with lower static priority. Requests for web pages with the same number of embedded objects are served in first-in-first-out order. Requests for the main HTML file and for the embedded objects in the same page are given the same priority level. This is achieved by using the real-time policy *SCHED_FIFO* of Linux.

- ♦ LNEF_LPH - is the LNEF policy in which a *lower* priority is given to the request for the main *HTML* file in comparison to the request for an embedded object in a given web page class.

As in the case of the regular LNEF policy, this policy gives higher priority to the requests for web pages with lower number of embedded objects. That is, requests associated with the HTML file and the embedded objects for a web page class WP1 will have a higher priority than those associated with the HTML files and embedded objects for a web page class WP2.

- ♦ LNEF_HPH - is the LNEF policy in which a *higher* priority is given to the request for the main *HTML* file in comparison to the request for an embedded object in a given web page class.

This is the opposite of LNEF_LPH policy. The request for the HTML file gets a higher priority than the requests for the embedded objects in the same page.

- ♦ Highest Number of Embedded objects First (HNEF)

This is the counterpart of LNEF. Higher priority is given to the requests for the web pages with more embedded objects. Requests for the main HTML file and for the embedded objects in the same web page are processed with the same priority level.

Although the relative performances of some of these policies are intuitive, it is important to understand the degree of performance improvement from using appropriate request characteristics in scheduling. Moreover, certain workload can lead to non-intuitive results (see Section 5.1.3 for example).

4.5 Performance Metrics

This section describes the performance metrics used in the experiments. They include the mean response time, mean slowdown and differentiation factor.

4.5.1 Mean Response Time

The response time of a web page request is the elapsed time measured in seconds from the moment that the *first* request for the main HTML document of a web page is issued to the moment that the reception of the *last* embedded objects at the client. Response time is directly perceived by a client. A long response time could mean an abortion of the request in the middle, which could waste server resources and hurt a web page owner's credibility. Therefore, the latency for an entire web page retrieval is an important performance metric.

Response time consists of a queuing delay on the server, the processing time and the transmission delay over the network. This value is influenced by factors such as the requested arrival rate, the size of files requested, the speed at which requests are processed, and the transmission speed of the network. The size of file and the data transmission delay due to network latency are negligible compared to the service demands of the requests to the server. Thus, the arrival rate, the service demand at the servers and the adopted scheduling policies are expected to have a major impact on the mean response time.

The mean response time is directly returned by `httpperf`. The number of runs is sufficient to guarantee a confidence interval of less than $\pm 5\%$ at a confidence level

of 95%. This is appropriate for comparing the performances of scheduling policies used at the back-end servers and the dispatcher.

4.5.2 Mean Slowdown

Mean slowdown [16, 23] is the ratio of the mean response time to the optimal mean response time. The optimal mean response time represents the optimal response time when each web page of a workload is the sole request in the system, ignorant of the overhead that occurs during processing and transmission. This metric indicates how far the server system operates from the optimal performance. The closer this value is to 1, the better the system performs.

In a parallel system with four back-end servers, the optimal mean response time is the mean service demand when the embedded objects of each web page can be served in parallel. Consider for example the workload with the following characteristics.

WP1($E=1$, $D=\text{fixed}$, $S=100\text{ms}$),

WP2($E=10$, $D=\text{fixed}$, $S=100\text{ms}$),

$\text{Pr}[\text{WP1}]=95\%$, $\text{Pr}[\text{WP2}]=5\%$.

The optimal response time for a request for WP1 is 200ms (100ms for processing the main HTML file plus 100ms for processing the embedded object) and the optimal response time for a request for WP2 is 400ms (100ms for processing the main HTML file + 300ms for processing embedded objects when they are served in

parallel in the four-backed-end-server system). Thus the optimal mean response time of the workload is 210ms ($0.95 \cdot 200 + 0.05 \cdot 400$).

4.5.3 Differentiation Factor

Differentiation Factor (DF) estimates the percentage gap between the mean response time of a scheduling policy and the mean response time of FCFS. Specifically, for a scheduling policy X:

$$DF = (\text{mean response time of X} / \text{mean response time of FCFS}) - 1$$

A policy performs better than FCFS if $DF < 0$. Since we are exploring the performance differences between priority-based policies and FCFS that is a non-priority-based policy, this performance metric is useful.

4.6 Validation of the Prototype

A number of validation experiments are conducted. The validation consists in comparing the measured performance metrics with expected system performance. As shown in the following subsection, the measured performance metrics agree with the expected results.

4.6.1 Applying Little's Law

Little's Law is one of most widely used theorems in queuing theory. It allows us to relate the mean number of jobs in the system with the mean response time. In our case, it treats the server system as a black box. In a stable system, the arrival rate

should be equal to the server throughput. Thus, the number of requests entering the system should be equal to those completing the services.

Little's Law states: $N = X \cdot R$, where N is the average number of requests in the system during the test duration, X is the number of HTTP GETs completed per second (GETs/sec) on the dispatcher and R is the mean response time per request. This validation experiment consists in measuring N and $X \cdot R$ and verifying that the values are indeed close to one another.

N is equal to the sum of response times for all requests divided by the test duration. In this thesis, N and R can be obtained from the client side, and X is obtained by keeping a counter of HTTP GETs on the dispatcher. Table 4-3 lists the percentage differences for N and $X \cdot R$ with various arrival rates for scheduling policy LNEF. Other scheduling policies result in similar results and are therefore not included. The validation results reported in Table 4-4 correspond to a system characterized by the following parameters: WP1($E=1$, $D=\text{fixed}$, $S=100\text{ms}$), WP2($E=10$, $D=\text{fixed}$, $S=100\text{ms}$), $\text{Pr}[\text{WP1}] = 95\%$, $\text{Pr}[\text{WP2}] = 5\%$. These results are based on 3000 web page requests.

Table 4-4: Little's Law Verification

λ	N	$X \cdot R$	Difference
4.6	1.43	1.44	0.3%
4.8	1.5	1.51	0.3%
5	1.58	1.58	0%
5.2	1.65	1.66	0.3%

It shows the negligibly small difference between the mean values of N and $X \cdot R$.

4.6.2 Arrival Rate vs. Mean Response Time

Because of the lack of queuing at a very low arrival rate, we expect that the mean response time from each type of scheduling policy should be comparable to the manually computed optimal mean response time. In section 4.5.2, we discuss the computation of the optimal mean response time of 210ms in a four-back-end-server system when the workload consists of 95% of the requests for web pages with 1 embedded object and 5% of the requests for web pages with 10 embedded objects. The measured response time is expected to be slightly higher than the calculated 210ms because of the system overheads and the small queuing that may take place from time to time.

Table 4-5 shows the measured performances of each policy and their percentage differences from the optimal mean response time of 0.21 seconds. The workload used in this validation experiment is characterized by the following parameters: $\lambda=1$ req/sec, WP1(E=1, D=fixed, S=100ms), WP2(E=10, D=fixed, S=100ms), $\text{Pr}[\text{WP1}] = 95\%$, $\text{Pr}[\text{WP2}] = 5\%$. These results are based on 14000 web page requests. The length of the experiment is the same as the length of the experiments used in Chapter 5.

Table 4-5: Mean Response Time at a Very Low Arrival Rate

Policies	FCFS	LNEF	HNEF
Mean Response Time (sec)	0.22	0.22	0.22
% Difference with Optimal	5%	5%	5%

After rounding the results to the nearest two decimal points, the performances of different policies are not distinct from each other. The discrepancy from the theoretical optimal is small.

4.6.3 Performances of High vs. Low Priority Web Pages

In a workload that consists of multiple web page classes, we expect a large performance gap between high and low priority pages. Specifically, with LNEF policy, the web page with fewer numbers of embedded objects should perform much better than the web page with more embedded objects; it is opposite for HNEF. The performance of both web page classes for FCFS is expected to sit in the middle of LNEF and HNEF.

This expectation can be used to verify the correctness of the system. Figure 4-4 and Figure 4-5 delineate, in the experiment, the performance of the web page classes with low and high number of embedded objects, respectively. The parameters used in this validation experiment are: WP1($E=1$, $D=\text{exponential}$, $S=100\text{ms}$), WP2($E=10$, $D=\text{exponential}$, $S=100\text{ms}$), $\text{Pr}[\text{WP1}] = 95\%$, $\text{Pr}[\text{WP2}] = 5\%$

Figure 4-4 graphs the response time of requests for WP1. As expected, we observe that the difference among the performances of the policies increases with an increase in the arrival rate. In this figure, WP1 of HNEF and LNEF perform the worst and the best, respectively.

Figure 4-4: Performance of Requests for a Web Page Class with a Fewer Numbers of Embedded Objects

Figure 4-5: Performance of Requests for a Web Page Class with a Larger Number of Embedded Objects

Figure 4-5 illustrates the performance of the requests for WP2. The performance of WP2 of HNEF and LNEF is the reverse of Figure 4-4, as we had previously indicated.

4.6.4 User-specified and Measured Arrival Rates

Httpperf needs a user-specified arrival rate for web page requests. The actual inter-arrival time of sending a request to the dispatcher is calculated after time-stamping each arrival time. All inter-arrivals are averaged at the end and their standard deviation is also calculated. The arrival rate is the inverse of the inter-arrival time. The specified arrival rate is then compared with the measured arrival rate. For all experiments conducted, no differences of specified and measured arrival rates occur after rounding the numbers to the nearest decimal points. The measured standard deviation is also well matched with the mean value. Note that the mean and standard deviation for an experimental inter-arrival time are expected to be equal to one another.

4.7 Implementation Issues

One possible way to implement the server complex is to use a single cluster for the servers and the dispatcher. The scheduling policies such as LNEF are based on web page characteristics. Information on the number of embedded objects in each web page can be stored in a lookup table placed in the dispatcher. Whenever a web

page is changed, the new number of embedded objects is to be sent to the dispatcher. Upon arrival of a request, the dispatcher can use the lookup table to determine the priority of the request. Such a lookup table based approach is expected to give rise to a small overhead when the dispatcher and the servers are in the same cluster. In case of LNEF_LPH, for example, the priority will also depend on the type of the requested component (the main HTML file or an embedded object).

Caching was turned off in our experiments for focusing on the performance impact of the scheduling policies. A real web server is likely to use caching for improving performance. The results of this research are still useful in such a situation. The caching of documents will be translated into a lower service demand associated with the requests. As a result, the saturation of the web server will be postponed to a higher arrival rate. Thus the performance differences among the scheduling policies observed on the experimental system are likely to appear at higher arrival rates on the real system.

Chapter 5 Results of Experiments

This chapter describes the results of experiments that compare the performances of the different scheduling policies. Section 5.1 studies the effects of various workload characteristics on performances. The impact of the number of back-end servers on performance is investigated in Section 5.2. The influence of using different priorities for the requests for the main HTML file and the embedded objects within the same web page on performance is explored in Section 5.3.

5.1 Impact of Workload Characteristics

The workload characteristics in this study include the arrival rate, the service demand, the number of embedded objects in a web page and the popularity of a web page class. These characteristics are expected to have a significant impact on performance.

5.1.1 Impact of the Arrival Rate

The performance of FCFS, LNEF and HNEF for various arrival rates are depicted in Figure 5-1 and Figure 5-2. The results are based on requests with fixed service demand and exponential service demand with the same mean.

Figure 5-1: Impact of Arrival Rate for Fixed Service Demand

Figure 5-2: Impact of Arrival Rate for Exponential Service Demand

As shown in Figure 5-1 and Figure 5-2, the distribution of the service demand only affects the absolute values of the mean response time, but not the relative performances of the scheduling policies.

No performance differences at the very low arrival rates, such as at 1.7 and 5.7 req/sec, are observed; the performance gap gradually widens with the increases in the arrival rates.

At the low arrival rates, due to the lack of queuing, requests are immediately served. Priority-based policies do not exert a noticeable influence at this time. The impact of priority-based policies is increasingly obvious when more and more requests are waiting for services. The underlined scheduling policies affect the queue length by manipulating the order of the services. The impact is high at a very high load, in which the queuing delay becomes the major component of the mean response time.

The LNEF policy is similar, although not identical, to the shortest-job-first policy in the operating system. The requests within a web page containing the least numbers of embedded objects are served first, which leads to a low overall mean response time. The difference from the traditional shortest-job-first policy arises from the fact that the main HTML file and the embedded objects of a web page are each served as separate requests. The parallel system allows the simultaneous processing of the embedded objects within a web page on different back-end servers.

In contrast to the LNEF, HNEF represents the counterpart of the longest-job-first policy. Requests for a web page that contains the larger number of embedded objects are processed with a higher priority. This policy leads to a large mean queuing

delay and demonstrates the worst performance (see Figure 5-1 and Figure 5-2 for example).

FCFS is a neutral policy and its performance lies between LNEF and HNEF.

The mean response time of the exponential service demand is usually higher than the mean response time of the fixed service demand, and the gap enlarges with an increase in arrival rates. Table 5-1 demonstrates the percentage increases of the mean response time achieved with an exponential service demand over that achieved with a fixed service demand.

Table 5-1: The Comparison of Fixed vs. Exponential Service Demand

Arrival Rate	FCFS	LNEF	HNEF
1.7	0%	0%	0%
5.7	8%	8%	11%
9.7	29%	28%	20%
13.7	94%	74%	95%
WP1(E=1, D=fixed, exponential, S=100ms), WP2(E=10, D=fixed, exponential, S=100ms), Pr[WP1] =95%, Pr[WP2] = 5%			

The exponential distribution leads to a larger variation of the service demand of each request. Table 5-1 shows that the performance gaps between two distributions increase with the rises of the arrival rates. Under a high load, the performance of the exponential service demand could be 90% worse than its fixed counterparts. Thus, it suggests that a large variability in service demand for the web page components can worsen the overall performance.

Exactly how significantly are LNEF and HNEF different from the neutral policy FCFS? Table 5-2 and Table 5-3 answer this question with the differentiation factors (DF). DF describes the percentage gap of LNEF or HNEF over FCFS. A negative number means a better performance than FCFS, and a positive number indicates a worse performance in comparison to FCFS.

Table 5-2: Impact of the Arrival Rate on Differentiation Factor for Fixed Service Demand

Arrival Rate	LNEF	HNEF
1.7	0%	0%
5.7	-4%	11%
9.7	-16%	23%
13.7	-49%	91%
WP1(E=1, D=fixed, S=100ms), WP2(E=10, D=fixed, S=100ms), Pr[WP1] =95%, Pr[WP2] = 5%		

Table 5-3: Impact of the Arrival Rate on Differentiation Factor for Exponential Service Demand

Arrival Rate	LNEF	HNEF
1.7	0%	0%
5.7	-4%	8%
9.7	-15%	32%
13.7	-43%	90%
WP1(E=1, D=exponential, S=100ms), WP2(E=10, D=exponential, S=100ms), Pr[WP1] =95%, Pr[WP2] = 5%		

From both tables, we notice that the performance advantage of LNEF over FCFS increases with an increase in arrival rates and it reaches almost 50% at a high load. HNEF, on the other hand, produces a mean response time that is 91% higher than that of FCFS. The results show that scheduling policies based on the number of embedded objects in a web page have a strong impact on performance.

A system manager might be concerned about how much higher the mean response time of a policy might be compared to an optimal value. Slowdown is a measurement that compares the mean response time of a policy with the optimal mean response time. As described in section 4.5.2, the optimal mean response time of a four-back-end-server system is 0.21 seconds for the fixed service demand of 100ms. Figure 5-3 graphs the impact of arrival rate on slowdowns.

Figure 5-3: Impact of Arrival Rate on Mean Slowdown

All policies can achieve the best slowdown value of close to 1 at the low arrival rates. LNEF demonstrates slowdown of 3 at a high load. The slowdown for HNEF and FCFS are 11 and 6 at the same high load. This implies that LNEF can sustain a high system load without a large degree of performance degradation.

5.1.2 Impact of Service Demand

Service demand of a back-end server corresponds to the simulated service time for a disk access and CPU time associated with the processing of an embedded object or the main HTML file. The impact of service demand on the performance of LNEF is captured in Figure 5-4. The service demand demonstrates similar impacts on the performance of FCFS and LNEF (see Appendix B).

Figure 5-4: Impact of Service Demand on the Performance of LNEF

The differences in response times achieved with the different service demands are almost the same for the arrival rate of 1.7, 3.7 and 5.7 req/sec. Starting from the arrival rate of 7.7 req/sec, the response time achieved with $S=150\text{ms}$ starts increasing more rapidly in comparison to the response times achieved with the other service demands.

Due to the shortage of queuing at the low arrival rates, the increase of response time mainly arises from the additional processing time on the server. Therefore, 50ms of additional service demand per request tends to produce an equal amount of the increase in the response time in both cases. However, the same amount of increase in the service demand produces a larger impact on a busier system than on a lightly loaded system.

Figure 5-5 compares the responsiveness of each policy to the rise of the service demand at $\lambda=9.7$ req/sec.

Figure 5-5: Impact of Service Demand on the Performances of Scheduling Policies

All policies show a higher performance degradation when the service demand is boosted from 100ms to 150ms than from 50ms to 100ms. The performance of FCFS worsens by 162% for the rise from 50ms to 100ms, and 529% is observed for the rise from 100ms to 150ms; the performance degradation increases from 123% to 317% for LNEF and from 200% to 895% for HNEF.

In addition, the performance gaps between LNEF and FCFS and between HNEF and FCFS widen with the increase in the service demand (see Figure 5-6).

Figure 5-6: Performance Comparison of Scheduling Policies for Different Service Demands

At S=50ms, differences are small. However, LNEF produces a response time that is 15% lower than FCFS at S=100ms and 43% lower than FCFS at S=150ms. HNEF produces a response time that is 32% higher than FCFS at S=100ms and 109%

higher than FCFS at $S=150\text{ms}$. This implies that the queuing delay experienced by HNEF is more pronounced than the other two policies; thus it exhibits the fastest growth of the mean response time, especially when the service demand leaps from 100ms to 150ms . LNEF is the least sensitive to the change of the service demand, and FCFS lies in the middle.

5.1.3 Impact of the Number of Embedded Objects

This section demonstrates how the performance of each scheduling policy is affected by the number of embedded objects. Three workloads are used: WL1, WL2 and WL3. The characteristics of the three workloads are presented.

WL1: WP1($E=1$, $D=\text{fixed}$, $S=100\text{ms}$), WP2($E=10$, $D=\text{fixed}$, $S=100\text{ms}$),

$\text{Pr}[\text{WP1}]=95\%$, $\text{Pr}[\text{WP2}]=5\%$.

WL2: WP1($E=2$, $D=\text{fixed}$, $S=100\text{ms}$), WP2($E=10$, $D=\text{fixed}$, $S=100\text{ms}$),

$\text{Pr}[\text{WP1}]=95\%$, $\text{Pr}[\text{WP2}]=5\%$.

WL3: WP1($E=3$, $D=\text{fixed}$, $S=100\text{ms}$), WP2($E=10$, $D=\text{fixed}$, $S=100\text{ms}$),

$\text{Pr}[\text{WP1}]=95\%$, $\text{Pr}[\text{WP2}]=5\%$.

The three workloads differ only in the number of embedded objects in WP1. Figure 5-7 depicts the performance of LNEF for the three workloads. Similar effects on FCFS and HNEF are observed (see Appendix C).

As shown, minimal variation of the mean response time is observed at the low arrival rate of 1.8 req/sec . The influence of the number of embedded objects becomes more pronounced at a higher arrival rate of 3.8 req/sec , and the impact escalates with

the rise of the arrival rate. Moreover, increasing the number of embedded objects when the system is at a higher load produces larger impacts than when the system is at a smaller load. A sharper increase in response time is observed when the workload changes from WL2 to WL3 than from WL1 to WL2. For instance, at the arrival rate of 7.8 req/sec, there is a 74% increase in the mean response time upon increasing one embedded object in a web page, in contrast to more than tripling the mean response time that occurs upon increasing another embedded object to the web page. At high loads, additional requests tend to overburden the server and the effect of increasing the number of embedded objects is more pronounced.

Figure 5-7: Impact of the Number of Embedded Objects on the Performance of LNEF

Figure 5-8 presents the effect of increasing the number of embedded objects in a web page on the performance of scheduling.

Figure 5-8: Impact of the Number of Embedded Objects on the Performances of Scheduling Policies

All three policies demonstrate a higher growth in mean response time with an increase in the number of embedded objects when the workload is changed from WL2 to WL3 in comparison to the change from WL1 to WL2. HNEF seems to be more sensitive to the change in the number of embedded objects than the other two policies.

LNEF's mean response time grows faster in comparison to FCFS. For WL2 and WL3, LNEF starts to demonstrate a performance that is comparable to that of FCFS (see Figure 5-9). This implies that the performance benefits of LNEF over FCFS decay at workloads containing a larger number the embedded objects.

Figure 5-9: Performance Comparison of Scheduling Policies for Different Number of Embedded Objects

Why is the increase of the number of embedded objects so influential? The reason is as follows. Competition for service exists not only among requests with different priorities, but also among requests with the same priority. We will discuss the effect of each of them on performance.

LNEF discriminates against pages with a larger number of embedded objects. This is captured in Table 5-4 that separately presents the mean response time achieved by the two classes of requests: one that is for WP1 (3 embedded objects) and the other that is for WP2 (10 embedded objects). The mean response time achieved at $\lambda=7.9$ req/sec for WP1 and WP2 are 2.38 seconds and 4.59 seconds

respectively. The response times produced by LNEF at the same arrival rate are 1.24 seconds and 31.4 seconds for WP1 and WP2 respectively. In comparison to FCFS, LNEF reduces the response time for WP1 by 48%. The corresponding increase in the response time for WP2 is disproportionately large: 584%. As a result, the overall performance advantage of LNEF over FCFS is reduced.

Table 5-4: Performance of Requests for Web Page Classes Achieved with FCFS and LNEF

Arrival Rates (req/sec)	Mean Response Time for FCFS (sec)		Mean Response Time for LNEF (sec)	
	WP1	WP2	WP1	WP2
1.9	0.24	0.53	0.26	0.59
3.9	0.29	0.65	0.33	0.88
5.9	0.43	0.95	0.49	1.84
7.9	2.38	4.59	1.24	31.41
WP1(E=3, D=fixed, S=100ms), WP2(E=10, D=fixed, S=100ms), Pr[WP1] = 95%, Pr[WP2] = 5%				

Competition among requests with the same priority is discussed next. If the requests with equal priority abound in the system, there is a lower chance that the embedded objects within a web page belonging to WP1 can be served in parallel. In case of WL1, serving an embedded object request for WP1 completes a web page request. However for WL3, requests for three embedded objects need to be served for each web page. Due to a reduction in concurrency, the processing of the objects may take a longer time to complete. As a result, the number of incomplete page requests is increased, as the workload is changed from WL1 to WL3 for example.

Increasing the number of the embedded objects in WP1 also raises the total system load substantially, since WP1 is selected by 95% of the requests. Both FCFS and LNEF experience such an increase of workload and their response times at a given arrival rate deteriorate as the number of embedded objects in WP1 is increased. Moreover, LNEF experiences the combined effects of the competition among web page classes with different priorities and the competition among the requests with the same priority. The overall effect is a deterioration in the relative performance of LNEF in comparison to FCFS. These issues are discussed further in Section 5.3, in which different priorities are given to requests for the main HTML file and the embedded objects in the same web page.

5.1.4 Impact of Web Page Class Popularity

The popularity of a web page class is denoted by the workload parameter $\text{Pr}[\text{WP}_i]$. A higher probability means a higher popularity of a web page class.

Figure 5-10 presents the effect of popularity on the performance of LNEF. Three different popularities of the web page class with a fewer numbers of embedded objects given by $\text{Pr}[\text{WP}_1]=90\%$, $\text{Pr}[\text{WP}_1]=95\%$ and $\text{Pr}[\text{WP}_1]=99\%$ are investigated. It implies that the popularity of the web page class with a higher number of embedded objects will be 10%, 5% and 1%, respectively. Similar trends are observed with FCFS and HNEF (see Appendix D).

When the popularity of the web page class with a smaller number of embedded objects increases, the total system load for a given arrival rate is reduced

since the workload contains fewer number of requests for the web pages with a larger number of embedded objects. From Figure 5-10, we observe that at a low system load, such as at the arrival rate of 3.2 and 6.2 req/sec, the workload reduction does not exert a significant impact on the performance. Evident influence starts to take place at a higher rate of 9.2 req/sec and the impact is appreciable at 12.2 req/sec.

Figure 5-10: Impact of Web Page Class Popularity on the Performance of LNEF

Figure 5-11 presents the degree of benefit that each scheduling policy experiences when WP1 becomes increasingly popular.

Like LNEF, FCFS and HNEF also produce large benefits with initial 5% rise (from 90% to 95%) of the popularity of WP1. The performance gain is smaller when Pr[WP1] changes from 95% to 99%. For FCFS, the performance improvement arising from the initial 5% increase in popularity is 68%, and it reduces to 49% from a

further 4% increase in popularity. The improvement experienced by LNEF is 70% and 41% for a change in $\text{Pr}[\text{WP1}]$ by 5% and 4% respectively. HNEF demonstrates performance improvements of 80% and 58% for a change in $\text{Pr}[\text{WP1}]$ by 5% and 4% respectively. Among the three, HNEF seems to be the most sensitive and LNEF is the least sensitive to a change in popularity.

Figure 5-11: Impact of Web Page Class Popularity on the Performances of Scheduling Policies

The performance differences among the three policies are gradually reduced as $\text{Pr}[\text{WP1}]$ increases from 90% to 95% and to 99% (see Figure 5-12). For example, the performance benefit experienced by LNEF over FCFS is 24% at the 90% popularity level, and it decreases to slightly 3% at 99% level. At high values of $\text{Pr}[\text{WP1}]$, such as 99%, most requests are for the same web page class and the

advantage of using scheduling policies based on the number of web objects is reduced.

Figure 5-12: Performance Comparison of Scheduling Policies for Different Web Page Class Popularities

5.2 Impact of the Number of Back-end Servers

In order to understand the impact of additional resources, such as a higher number of back-end servers, on the performances of the scheduling policies, a set of experiments are conducted for 2, 3 and 4 back-end servers.

Figure 5-13 shows the effect of increasing the number of back-end servers on the performance of LNEF. Similar results are observed for FCFS and HNEF (see Appendix E).

Figure 5-13: Impact of the Number of Back-end Servers on the Performance of LNEF

At very low arrival rates, no significant impact is observed for using additional resources, since the servers are free most of the time and extra processing power does not make much difference in performance. At a high arrival rate, contention for resource starts occurring and the importance of using extra servers is increased. For example, a large improvement in performance is observed at $\lambda=7$ req/sec when B is increased from 2 to 3.

Nevertheless, the efficiency of incorporating an additional server gradually decays. The performance improvement of adding one extra back-end server from 3 to 4 is less effective than adding one more back-end server when B is equal to 2. At the arrival rate of 7 req/sec, the improvement in mean response time is 81% when the

number of back-end servers rises from 2 to 3, versus 16% when the number of back-end servers is increased from 3 to 4.

Figure 5-14 describes the impact of the additional servers on the performances of the different scheduling policies.

Figure 5-14: Impact of the Number of Back-end Servers on the Performances of Scheduling Policies

A sharp decrease in mean response time is observed for all of the scheduling policies when the number of back-end servers is up from 2 to 3. The performance of FCFS is enhanced by 88% when increasing B from 2 to 3 and the improvement is 27% for incorporating one more server. For HNEF, the improvements are 94% and 35%, respectively. LNEF displays a lesser sensitivity to the increase in the number of back-end servers.

At the given arrival rate of 7 requests/sec, the performance superiority of FCFS over HNEF and the superiority of LNEF over FCFS diminish when the number of the back-end servers is increased (see Figure 5-15). At $B=2$, performance of LNEF is 50% better than FCFS, and the improvement is reduced to 4% at $B=4$. The mean response time for HNEF is almost 4 times worse in comparison to FCFS at $B=2$, and it decreases to a value that is 15% worse than the mean response time for FCFS at $B=4$.

Figure 5-15: Performance Comparison of Scheduling Policies for Different Number of Back-end Servers

5.3 Priority for Web Page Components

This section describes the effect of giving different priorities for the main HTML file and the embedded objects of a web page. Unlike the regular LNEF, in

which the HTML file and the embedded objects of a given web page are processed at the same priority level, LNEF_LPH gives a lower priority to the main HTML file in comparison to the embedded objects of the same web page; LNEF_HPH, on the other hand, processes the HTML file at a higher priority than the embedded objects of the same web page. As in the case of LNEF, both the priorities of the HTML file and the embedded objects of a page with a smaller number of embedded objects are higher than the priorities for the main HTML file and the embedded objects of a page with a higher number of embedded objects.

Figure 5-16 displays the performances of the three policies.

Figure 5-16: Impact of Using Priority for Web Page Components

Since the system is heavily used at high arrival rates, the performance differences are pronounced at the higher arrival rates. At the arrival rate of 14 req/sec,

the mean response time for LNEH_LPH is 19% lower than that for LNEF, and the mean response time for LNEF_HPH is 15% worse than that for LNEF. As mentioned in Section 5.1.3, for regular LNEF, competitions exist not only among requests for different classes of web pages, but also among requests for the components of a given web pages class.

The competition among requests for components of the same web page class is discussed first. The degree of parallelism achieved for processing the embedded objects within the same web page are different for LNEF, LNEF_LPH and LNEF_HPH. Using the LNEF_LPH policy, the embedded objects in a web page have the highest possibility of being served in parallel; on the other hand, LNEF_HPH policy demonstrates the lowest possibility of parallel processing and LNEF sits in the middle. A brief explanation for this behavior is provided in the next paragraph.

For LNEF_LPH, since the HTML file is given a lower priority than the embedded objects within the same web page, the HTML file of a new-coming web page tends to be kept waiting while the embedded objects of the existing requests are served concurrently. Since the processing of the HTML file is delayed, the embedded objects of this new web page will be delayed from entering the system. With a fewer number of requests in the system, the embedded objects of a web page have a higher chance of being served concurrently.

However, with the LNEF_HPH policy, the embedded objects of a web page are less likely to be served in parallel. The HTML file of a new-coming web page has a higher priority than the embedded objects of the web pages waiting in the system. Once the new HTML file is processed, requests for its embedded objects will also

start contending for resources. Thus, LNEF_HPH tends to increase the number of unfinished requests in the system. As a result, more web page requests exist in the system and the degree of parallelism for processing a given web page decreases.

As indicated in Section 5.1.3, the advantage of LNEF over FCFS can decay when more embedded objects are contained in a web page (see Figure 5-9). LNEF_LPH can overcome such a shortcoming of the pure LNEF policy. Figure 5-17 (a) plots the performance of FCFS, LNEF and LNEF_LPH for the workload in which WP1 contains 2 embedded objects and WP2 contains 10 embedded objects.

At the arrival rate of 10.1 req/sec, LNEF has a gain of 8% over FCFS, while LNEF_LPH performs better than LNEF by 35%.

Similar benefits in using LNEF_LPH can be observed for the workload that consists of a web page class with 3 embedded objects and a web page class with 10 embedded objects (see Figure 5-17 (b)).

As mentioned in Section 5.1.3, the discrimination against WP2 requests introduced by LNEF is also responsible for its inferior performance for a higher number of embedded objects in WP1. For a $\lambda = 7.9$ req/sec, the response time for WP1 and WP2 achieved by LNEF_LPH are 0.77 seconds and 20.39 seconds respectively. The discrimination against WP2 performed by LNEF_LPH seems to be smaller in comparison to the discrimination introduced by the regular LNEF (see last row of Table 5-4).

(a)

(b)

Figure 5-17: Performance of LNEF_LPH (a) $E_1=2$ (b) $E_1=3$

Chapter 6 Conclusions

Section 6.1 provides a summary of the work undertaken in this thesis. Section 6.2 presents the conclusions that are drawn based on the experimental results. Section 6.3 offers directions for the future research.

6.1 Summary

In order to enhance the web server performance in a heavy traffic environment, this thesis investigates the possibility of achieving a high performance through priority-based scheduling policies. The performances of a neutral policy and priority-based policies are studied. Requests are prioritized according to the number of embedded objects in a web page. The policies studied include: First Come First Served, Lowest Number of Embedded Objects First, Highest Number of Embedded Objects First, a LNEF policy that gives a lower priority to the main HTML file in comparison to the embedded objects of a web page and a LNEF policy that gives a higher priority to the main HTML file in comparison to the embedded objects of a web page.

The performance prototype is based on the popular open source Apache web server running on a network of Pentium PCs under the Linux operating system. The system is a three-tier system that consists of clients, a dispatcher and multiple back-end servers. The synthetic workload and the measurement-based approach is used

since it can capture the effect of system overheads that could be difficult to capture accurately in a pure simulation or analytic model-based investigation. Through this performance prototype, the research analyzes the effect of various scheduling policies on system performance.

Httpperf is a tool developed by Hewlett-Packard to simulate the workload produced by a client browser and is used in the performance measurements for the experiments. Httpperf reads web page requests from a user-defined input file. Each web page is treated in a session by httpperf. Each session consists of a request for the main HTML file and multiple requests for the embedded objects. The web page requests are sent to the server system according to the user-specified arrival rate. We have used an open model so that a continuous flow of requests is sent to the system until the experiment stops. Modifications of httpperf are also undertaken to extract more relevant performance information from the tool. The dispatcher and multiple back-end servers are reengineered Apache web servers. The dispatcher is configured into a reverse proxy through the rewrite module.

Through comparing the performances of various scheduling policies and exploring the effects of various workload characteristics on the performance, we derive valuable insights into web server scheduling that are presented in the next section.

6.2 Conclusions

Compared to the neutral policy FCFS, giving higher priority to the web pages with the lowest number of embedded objects (the LNEF policy) achieves a substantial

performance benefit. On the other hand, giving higher priority to the web pages with the highest number of embedded objects (the HNEF policy) degrades the performance even to a larger degree. The performance difference occurs primarily at a high system load. Scheduling policies such as LNEF are based on the knowledge of the number of embedded Objects in a web page. As described in Section 4.7, such a knowledge can be acquired without excessive overheads if the server complex is implemented on a single cluster for example.

The benefits of LNEF shrink with the added number of embedded objects in the smaller web page class WP1. The performance decay of LNEF at a higher number of embedded objects is due to a competition among requests with an equal priority as well as a competition among requests for different web page classes. As a result of the first, a large number of requests may be served at the same time. Hence, the embedded objects within the same web page are less likely to be served in parallel. The discrimination against requests for WP2 introduced by LNEF increases the response time of requests for WP2 in a disproportionate fashion. As a result, the performance benefit of LNEF is reduced. Giving different priorities to the main HTML document and the embedded objects of a web page can overcome this problem. Giving a lower priority to the main HTML file than the embedded objects (the LNEF_LPH policy) offers a significant benefit over regular LNEF and FCFS.

We have investigated the impact of variability in service demand on performance by using a fixed and an exponential distribution. A workload with a variation of service demand in web page components (the main HTML file or the embedded objects) gives rise to an inferior performance in comparison to a workload

with no variation of the service demand in components. This performance difference increases with the increase in system load.

All policies are more sensitive to the change of the mean service demand at a higher arrival rate. They react sharply to the additional service demand if the current level already creates a sufficiently high workload for the system. HNEF is more responsive to a variation in service demand than other policies and LNEF is the least responsive policy. The performance differences between HNEF and FCFS and between LNEF and FCFS are more observable at a high load.

The performance differences among the policies tend to diminish as the popularity for WP1, as captured in $\Pr[\text{WP1}]$, increases. HNEF is most sensitive to a change in popularity; LNEF is the least sensitive to such a change, and FCFS is placed in the middle.

The incorporation of additional back-end servers improves the performance sharply at first; however, the benefits at a given arrival rate gradually diminish as more and more servers are added.

The next section suggests directions for future research.

6.3 Future Research

Future research addresses three areas: priority-based scheduling policies, next generation of the Apache web server and real-time operating system.

Using a priority scheme such as LNEF may lead to the “starvation” of certain requests. The low priority requests might be kept being pushed to the end of the waiting queue when the system load is high. An aging technique [35] for boosting the

priority-of requests that have been waiting for a long time needs further research. The frequency of running such an aging algorithm should strike a compromise between the concomitant system overhead and response time improvement.

This research is concerned with systems in which data is fully replicated. Investigating the performance of scheduling policies on systems in which each web page component is replicated only on a subset of the servers warrants investigation.

Admission control can be combined with the use of the priority-based policies. A threshold value can be used to control the total number of requests entering the system. In addition, a second threshold value that limits the number of low priority requests entering the system can also be considered. The effects of setting the threshold values need to be investigated since the clients that request low priority web pages might never come back once their requests are rejected.

A hybrid policy that prioritizes requests according to the number of embedded objects in a web page and a client ID (to identify requests from different clients) can be studied. More specifically, requests for web pages with an equal number of embedded objects can be further prioritized based on the importance of the client that the request comes from.

Apache 2.0 which runs in a hybrid of thread/process model on the Linux operation system is now available. The priority-based scheduling policies supported by such web servers is expected to demonstrate a visible enhancement beyond the Apache 1.3.19 that has been used in the thesis. It will benefit from the scalability that the light-weighted threads can provide. Investigating performance based on such a system is another direction for future research.

Caching has been disabled at the dispatcher as well as at the back-end servers. In the presence of caching, the workload and service demand would be changed according to the cache hit ratio, which can affect overall system performance. Exploring the effect of caching on overall performance of a system that uses the priority-based scheduling policies can be studied in the future.

Real-time Linux is expected to overcome the shortcomings of the general Linux OS in undertaking real-time scheduling. Problems arising from paging delay and large context switching overheads would be minimized in real-time Linux. The performance benefits which would accrue from such a real-time operating system are worthy of future studies.

References

- [1] T. F. Abdelzaher, N. Bhatti, "Web Content Adaptation to Improve Server Overload Behavior," *Computer Networks*, Vol. 31, No. 11-16, May 1999, pp. 1563-1577.
- [2] J. Almeida, M. Dabu, A. Manikutty, P. Cao, "Providing Differentiated Levels of Service in Web Content Hosting," in *Proceedings of the ACM Workshop on Internet Server Performance (held in conjunction with the ACM SIGMETRICS'98 conference)*, Madison, USA, June 1998, pp. 91-102.
- [3] Apache Software Foundation, "Apache Web Server".
Available at: <http://www.apache.org>.
- [4] M. Aron, "Differentiated and Predicable Quality of Service in Web Server Systems," PhD Thesis, Department of Computer Science, Rice University, October 2002.
Available at: <http://citeseer.nj.nec.com/aron00differentiated.html>.
- [5] M. F. Arlitt, C. L. Williamson, "Web Server Workload Characterization: the Search for Invariants," in *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, USA, April 1996, pp. 126-137.
- [6] M. F. Arlitt, C. L. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications," *IEEE/ACM Transactions on Networking*, Vol. 5, No. 5, October 1997, pp. 631-644.
- [7] M. F. Arlitt, R. Friedrich, T. Jin, "Workload Characterization of a Web Proxy in a Cable Modem Environment," *ACM SIGMETRICS Performance Evaluation, Review*, Vol. 27, No.2, September 1999, pp. 25-36.
- [8] T. F. Abdelzaher, N. Bhatti, "Web Server QoS Management by Adaptive Content Delivery," in *Proceedings of the 7th International Workshop on Quality of Service*, London, England, May 1999, pp. 216-225.
- [9] G. Banga, P. Druschel, J. C. Mogul, "Resource Containers: A new facility for Resource Management in Server Systems," in *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, New Orleans, USA, February 1999, pp. 45-58.
- [10] N. Bhatti and R. Friedrich, "Web Server Support for Tiered Services," *IEEE Network*, Vol. 13, No. 5, September 1999, pp. 64-71.
- [11] D. P. Bovet, M. Cesati, "Understanding the Linux Kernel," O'Reilly and

Associates, 2001.

- [12] X. Chen, P. Mohapatra, H. Chen, "An Admission Control Scheme for Predictable Server Response Time for Web Accesses," in Proceedings of 10th International World Wide Web Conference, Hong Kong, May 2001, pp. 545-554.
- [13] L. Cherkasova, "Scheduling Strategy to Improve Response Time for Web Applications," in Proceedings of the High-Performance Computing and Networking Conference (HPCN Europe), Amsterdam, Netherlands, April 1998, pp. 305-314.
- [14] L. Cherkasova, P. Phaal, "Session Based Admission Control: a Mechanism for Improving the Performance of an Overloaded Web Server," Technical Report, Hewlett Packard, 1998.
Available at: <http://www.hpl.hp.com/techreports/98/HPL-98-119.html> or <http://citeseer.nj.nec.com/cherkasova98session.html>.
- [15] M. E. Crovella, A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," IEEE/ACM Transactions on Networking, Vol. 5, No. 6, December 1997, pp. 835-846.
- [16] M. E. Crovella, R. Frangioso, M. Harchol-Balter. "Connection Scheduling in Web Servers," in Proceedings of USENIX Symposium on Internet Technologies and Systems, Boulder, USA, October 1999, pp. 243-254.
- [17] P. Druschel, G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems", in Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI), Seattle, USA, October 1996, pp. 261-275.
- [18] L. Eggert, J. Heidemann, "Application-Level Differentiated Services for Web Servers," World Wide Web Journal, Vol. 32, No. 3, March 1994, pp. 70-81.
- [19] R. S. Engelschall, "A User Guide to URL Rewriting with the Apache Web Server", Apache group and FreeBSD Developer Team.
Available at: <http://www.engelschall.com/pw/apache/rewriteguide>.
- [20] R. S. Engelschall, "Load Balancing Your Web Site", Web Techniques Magazine, Vol. 3, No. 5, May 1998.
Available at: <http://www.webtechniques.com/archives/1998/05/engelschall>.
- [21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, "RFC 2068: Hypertext Transfer Protocol – HTTP/1.1," January 1997.
Available at: <http://rfc.sunsite.dk/rfc/rfc2068.html>.

- [22] D. Gaudet, "Apache Performance Notes," Developer of Apache Web Server, Available at: <http://www.apache.org/docs/misc/perf-tuning.html>.
- [23] M. Harchol-Balter, N. Bansal, B. Schroeder, M. Agrawal, "Implementation of SRPT Scheduling in Web Servers," in Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS), Goteborg, Sweden, July 2001, pp. 1-24.
- [24] A. Iyengar, Ed MacNair, T. Nguyen, "An Analysis of Web Server Performance," in Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM), Phoenix, USA, November 1997, pp. 1943-1948.
- [25] R. Jain, "The Art of Computer Systems Performance Analysis," John Wiley & Sons, Inc, 1991.
- [26] K. Li, S. Jamin, "Measurement-Based Admission-Controlled Web Server," in Proceedings of the IEEE Infocom Conference, Tel-Aviv, Israel, March 2000, pp. 651-659.
- [27] C. Ehrhardt, B. Deianov, "Linux-Kernel Archive: SCHED_RR is broken + patch," Linux-Kernel mail list, April 25, 2000.
Available at:
<http://www.uwsg.iu.edu/hypermail/linux/kernel/0004.3/0197.html>,
<http://www.uwsg.iu.edu/hypermail/linux/kernel/0004.3/0300.html>.
- [28] P. Li, "QoS-Enabled E-commerce Web Application," MCS Thesis, School of Computer Science, Carleton University, Ottawa, Canada, 2001.
- [29] S. Majumdar, E. Parsons, "Parallel Job Scheduling: a Performance Perspective," Performance Evaluation: Origins and Directions (eds: G. Haring, C. Lindemann, M. Reiser), Lecture Notes in Computer Science, Springer Verlag 2000, pp. 233-252.
- [30] J. Mogul, K. K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel," ACM Transaction of Computer Systems (TOCS), Vol. 15, No. 3, August 1997, pp. 217-252.
- [31] D. Mosberger and T. Jin, "httpperf: A Tool for Measuring Web Server Performance," in Proceedings of the 1st Workshop on Internet Server Performance (WISP), Madison, USA, June 1998, pp. 59-67.
- [32] S. Nadimpalli, S. Majumdar, "Performance Enhancement Techniques for Parallel Web Servers", in Proceedings of the IEEE International Conference on Parallel Processing (ICPP), Toronto, Canada, August 2000, pp. 233-241.

- [33] R. Pandey, J. Fritz Barnes, R. Olsson, "Supporting Quality of Service in HTTP Servers," in Proceedings of the 17th Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Puerto Vallarta, Mexico, June 1998, pp. 247-256.
- [34] W. R. Stevens, "Unix Network Programming," Vol. 1, 2nd Edition, Prentice Hall, 1997.
- [35] A. Silberschatz, P. Galvin, "Operating System Concepts," 5th Edition, John Wiley & Sons Inc., 1998.
- [36] W. Stevens, "RFC 2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," January 1997.
Available at: <http://www.faqs.org/rfcs/rfc2001.html>.
- [37] N. Vasiliou, "Overview of Internet QoS and Web Server QoS", Department of Computer Science, The University of Western Ontario, London, Canada, April 2000.
Available at: <http://citeseer.nj.nec.com/vasiliou00overview.html>.
- [38] T. Voigt, P. Ginningberg "Handling Persistent Connections in Overloaded Web Servers," Uppsala University, Sweden, September 2001.
Available at: <http://citeseer.nj.nec.com/467725.html>.
- [39] C. A. Waldspurger, and W. E. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management," in Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation (OSDI), Berkeley, USA, November 1994, pp. 1-11.

Appendix A

An example input file for the httpperf request generator

```
#session 1

/test1.html
  /pict6.gif
  /pict7.gif
  /pict8.gif
  /pict9.gif
  /pict10.gif
  /pict11.gif
  /pict12.gif
  /pict13.gif
  /pict14.gif
  /pict15.gif
```

```
#session 2

/test2.html
  /pict0.gif
```

This example input file specifies two sessions of http requests. Each session represents one web page request that consists of the request for the main HTML file and multiple requests for the embedded objects. The requests for the embedded objects are not started until the request for the main HTML file is answered. The requests for test1.html and test2.html are requested according to a user-specified arrival rate.

Appendix B

Figure B-1 and Figure B-2 demonstrate the impact of the service demand on the performances of FCFS and HNEF respectively.

Figure B-1: Impact of Service Demand on the Performance of FCFS

Figure B-2: Impact of Service Demand on the Performance of HNEF

Appendix C

Figure C-1 and Figure C-2 demonstrate the impact of the number of embedded objects on the performances of FCFS and HNEF respectively.

Figure C-1: Impact of the Number of the Embedded Objects on the Performance of FCFS

Figure C-2: Impact of the Number of the Embedded Objects on the Performance of HNEF

Appendix D

Figure D-1 and Figure D-2 demonstrate the impact of the web page class popularity on the performances of FCFS and HNEF respectively.

Figure D-1: Impact of the Web Page Class Popularity on the Performance of FCFS

Figure D-2: Impact of the Web Page Class Popularity on the Performance of HNEF

Appendix E

Figure E-1 and Figure E-2 demonstrate the impact of the number of back-end servers on the performances of FCFS and HNEF respectively.

Figure E-1: Impact of the Number of Back-end Servers on the Performance of FCFS

Figure E-2: Impact of the Number of Back-end Servers on the Performance of HNEF