January 2007

Performance Analysis of Offloading Application-Layer Tasks to Network Processors

Soumya Mahadevan
University of Massachusetts Amherst

PERFORMANCE ANALYSIS OF OFFLOADING APPLICATION-LAYER TASKS TO NETWORK PROCESSORS

A Thesis Presented

by

SOUMYA MAHADEVAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

September 2007

Electrical and Computer Engineering

PERFORMANCE ANALYSIS OF OFFLOADING APPLICATION-LAYER TASKS TO NETWORK PROCESSORS

A Thesis Presented

by

SOUMYA MAHADEVAN

Approved as to style and content by:

Tilman Wolf, Chair

Aura Ganz, Member

Russell Tessier, Member

C. V. Hollot, Department Head
Electrical and Computer Engineering

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Tilman Wolf, for his advice and guidance. His enthusiasm has always kept me motivated and his suggestions have been very useful during the course of the project. I would like to thank Netronome for the system components, and Jim Wasson and Robert Truesdell for their invaluable technical support. Thanks to the folks at NSL for the fun times during the last two years. Special thanks to my parents for their love and support.

ABSTRACT

PERFORMANCE ANALYSIS OF OFFLOADING APPLICATION-LAYER TASKS TO NETWORK PROCESSORS

SEPTEMBER 2007

SOUMYA MAHADEVAN

B.E., UNIVERSITY OF MUMBAI, INDIA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Tilman Wolf

Offloading tasks to a network processor is one of the important ways to increase server performance. Hardware offloading of Transmission Control Protocol/Internet Protocol (TCP/IP) intensive tasks is known to significantly improve performance. When the entire application is considered for offloading, the impact on the server can be significant because it significantly reduces the load on the server. The goal of this thesis is to consider such a system with application-level offloading, rather than hardware offloading, and gauge its performance benefits.

I am implementing this project on an Apache httpd server (running RedHat Linux), on a system that utilizes a co-located network processor system (IXP2855). The performance of the two implementations is measured using the SPECweb2005 benchmark, which is the accepted industry standard for evaluating Web server performance.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

The Internet is a communication infrastructure that interconnects a large community of end users and content providers. The key to the success of the Internet is the ease of usage on both sides. It appeals to end users because it is readily available and provides access to a large amount of data and services. It is simple for content providers to provide services via the Internet. By purchasing a network connection, a content provider can reach to an audience of hundreds of millions of users.

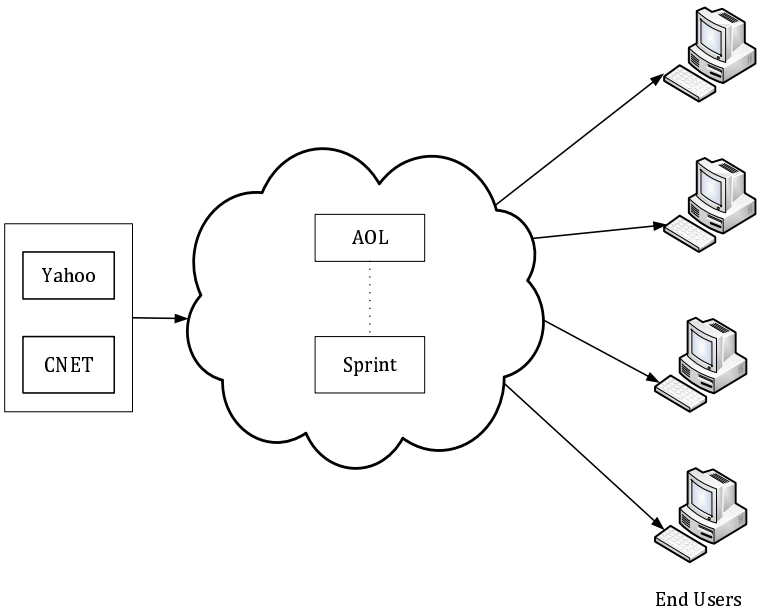

Figure 1.1. Connecting content providers to end users

The Internet offers content providers lots of opportunities to reach out to new audiences. However, the instability and unpredictability of the Internet's performance

can have a negative impact on the relationship between service providers and customers. Most Web users would abandon a site because of performance difficulties, given the wide range of service providers they get to choose from (Fig.1.2). A high performing website translates into customer satisfaction and increased market share. Therefore, it is critical to address performance bottlenecks in the Internet in order to overcome the challenges posed by it.

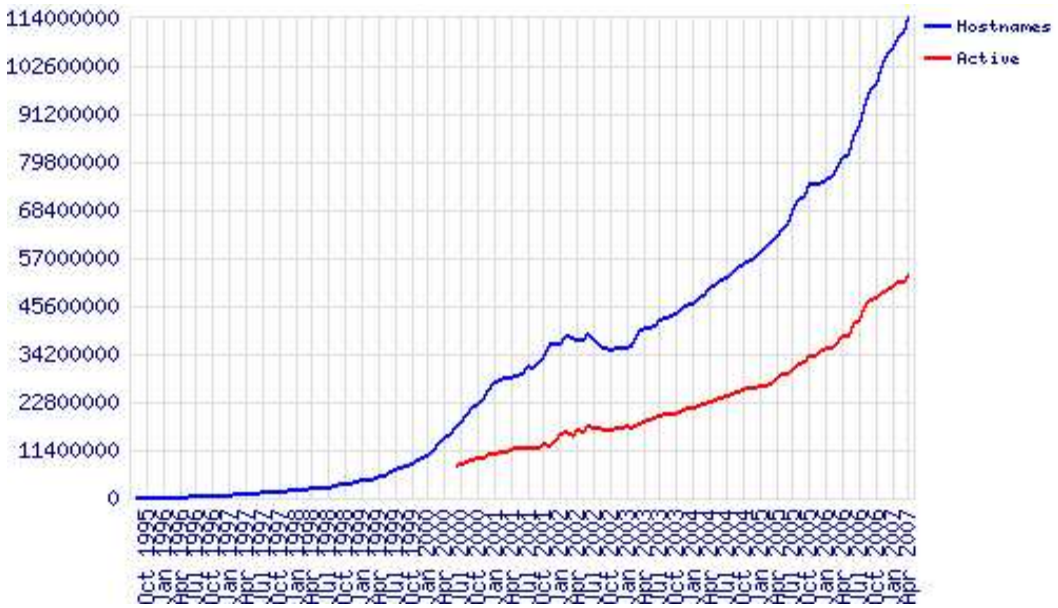

Figure 1.2. Domain name increase between 1997 to 2007

Courtesy: Netcraft (http://www.netcraft.com)

This thesis addresses one of the performance bottlenecks, that posed by Web server performance. It proposes an innovative way to deploy network processors to offload applications from Web servers, and thereby leaving the server available to respond to more client requests.

1.2 Internet Bottlenecks

The Internet is essentially a network of networks. In order for the Internet to function as a single global entity, all of its individual networks must connect to each

other and exchange information. This is done by establishing a peering session between a pair of routers, each located at the border of one of the two networks which need to exchange traffic. The routers exchange information that keep them updated on how to connect to reachable networks. Peering points and routing protocols thus connect the disparate networks of the Internet into one single entity. Connecting to one of these networks automatically provides access to all Internet users and servers. This structure of the Internet as an interconnection of individual networks is the key to its scalability. It enables distributed administration and control of all aspects of the system. However, there are a number of bottlenecks inherent in this structure, that can slow down performance and reduce the ability of the Internet to handle its ever increasing number of users and services:

1. **First mile**

   This bottleneck arises when a centralized model is used to serve content on the Internet. A content provider can set up his Web server in a single physical location. The speed at which Internet users can access information from it is necessarily limited by its First Mile connectivity - the bandwidth capacity of the website's connection to the Internet. Not only does the content provider need to buy larger connections to his or her ISP, the ISP also needs to expand its internal network capacity. Therefore, this centralized structure is inherently unscalable. This can be partially solved by adopting a distributed server cluster where servers are set up in different physical locations and can serve multiple requests.

2. **Peering points**

   This bottleneck occurs at the interconnection points between independent networks and is mostly of an economic nature. Networks do not have any profit gained by setting up free peering arrangements, nor is it suitable for them to pay

another network for peering. Therefore, the limited number of peering points between networks end up as bottlenecks.

3. **Backbone bandwidth**

   The structure of the Internet is such that all network traffic traverses one or more backbone networks. This means that the capacity of the backbone networks must be able to grow as quickly as Internet traffic. Fiber is cheap and able to support high bandwidth demands. However, the routers at the ends of the fiber cables limit capacity. The speed of the packet forwarding hardware and software in routers is limited by the current technology. Router capacity improvements have not kept pace with the increase in traffic. IP over ATM is an alternative to speedup the network, but is more expensive to deploy and maintain.

   The mismatch of demand and capacity of backbone bandwidth makes the backbone a serious bottleneck.

4. **Last Mile**

   The Internet is only as fast as its slowest link. However, when the last mile problem is solved, the bottleneck resulting from the remaining factors will be increasingly more apparent.

5. **Server performance**

   The World Wide Web has grown tremendously in the last decade in terms of users and volume of information. WWW traffic will dominate network traffic for the foreseeable future. This has placed substantial performance demands on Web servers. The server in any transaction needs to be able to handle the amount of computation required to deliver content in a responsive manner. It has defined load limits and it can serve only a certain maximum number of requests per second. However, the increased complexity of protocols have made

applications tend to be very computationally expensive on server performance. For example, the commonly deployed SSL protocol significantly slows down performance.

The goal of my thesis is to improve Web server performance, one of the important Internet bottlenecks. A number of approaches have been adopted to increase performance. Caching data on the main system or another system located near the clients can significantly reduce access times and server loads. TCP offloading onto a network interface is also commonly used and can significantly reduce computation requirements for packet processing on the host CPU. There have been many variations of this methodology. The number of offloaded connections can be reduced by offloading only a subset of TCP connections to the network interface, thereby preventing overload and performance degradation [23]. Implementation and advantages of network processors integrated with the host CPU more suited to high bandwidth TCP/IP networking are detailed in [2]. However, most of the approaches dealing with offloading focus solely on the TCP/IP stack (TOE).

In my thesis, I utilize a network processor for application-level offloading, rather than offloading only the network protocol stack. A reverse proxy runs on the network processor that acts as a static cache and serves some of the requests directed to the Web server. Many network intensive applications naturally perform better on network processors. Since these applications form a bulk of the requests serviced, appreciable performance benefits can be expected. I also provide a performance evaluation between this approach and the traditional host-only approach.

1.3 Summary of Chapters

My proposal has the following outline:

1. In Chapter 2, I provide a brief description of Web servers and proxy servers and some of their performance limitations. I also describe the efforts that have been made in improving different performance functionalities.

2. Chapter 3 explains the concept and motivation for application-level offloading. I provide an introduction to it using techniques that are already in use, such as TCP/IP offloading. I also describe the architectural setup of the system implemented. The chapter also discusses suitable designs and components for a prototype - namely the Network Processor (NP) for offloading, the Apache httpd as the Web server component and Squid as the reverse proxy.

3. Chapter 4 introduces the concept of benchmarking, and how it is used and the metrics it measures. I also explain some features of SPECweb2005, which is the benchmark I use. The most important details in this chapter are the performance measurements taken for the benchmark workload. The chapter also analyzes the metrics variations as detailed in the plots.

4. Finally, in Chapter 5, I provide a conclusion for my thesis. I also provide a list of improvements, which, if made to the system, would bring about a considerable improvement in performance.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Web Servers

2.1.1 Definition and Working

The HyperText Transfer Protocol (HTTP) [8] is the de facto standard for transferring WWW documents. It is a request/response protocol between clients and servers. The client program and the server program talk to each other exchanging HTTP messages. HTTP defines the structure of these messages and the methodology of exchange. HTTP operates over Transmission Control Protocol (TCP) [10] connections, usually to port 80, although this can be overridden.

A Web server implements the server side of HTTP. It can be defined as a computer that accepts HTTP requests from clients, and serves them HTTP responses, that are usually data, such as HTML documents and images. Popular Web servers include Apache, Microsoft Internet Information Server, Sun and Zeus.

HTTP defines how clients request Web pages from servers and how this information is exchanged. A brief description of the process is given below:

1. The HTTP client first initiates a TCP connection with the server. Port 80 is used as the default port at which the server listens for requests from HTTP clients.

2. The client sends an HTTP request message to the server through the socket associated with the established TCP connection.

3. The server receives the request through the socket associated with the TCP connection, retrieves the requested object and encapsulates the object within the HTTP response message and sends it back to the client. Depending on whether *persistent* or *non-persistent* connections are used, the server will either leave its connection open or close it.

4. The HTTP client receives the response and either terminates the connection or requests for a new object.

The RTT (Round Trip Time) can be improved using pipelining. The figure below shows HTTP in action:

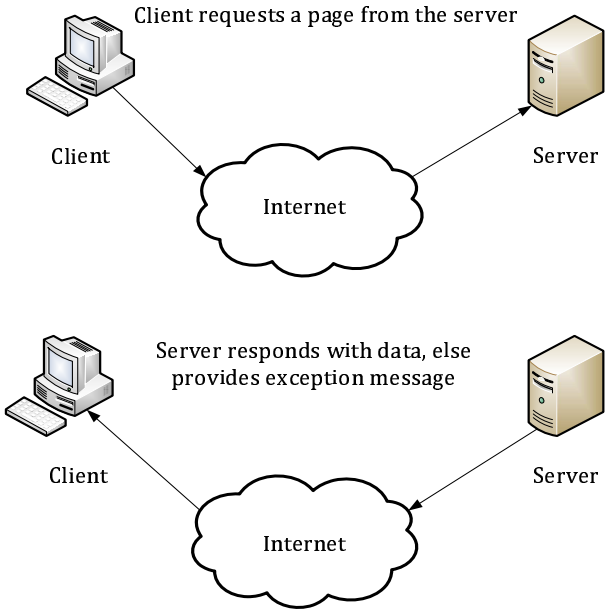

Figure 2.1. HTTP Request/Response between Web server and client

2.1.2 Performance Parameters

Web servers serve requests from many TCP connections at the same time. There are a number of performance parameters on which Web servers can be measured and compared. The following are quantitative estimates:

1. Number of requests that a server can handle per second.

2. Latency (in ms) for each request.

3. Throughput in bytes/second.

4. Concurrency level, which is essentially the number of computations that can execute overlapped in time, and permit sharing of common resources.

5. Scalability level

2.1.3 Limitations

Due to the dynamic nature of the Web, providing an acceptable quality of service (QoS) is a challenging task. Supporting QoS in servers has been extensively documented in [16]. The Web services QoS requirement refers to a variety of aspects including performance, reliability, integrity, accessibility, availability, interoperability, and security. Each of the parameters is discussed below, including their inherent limitations and some solutions to alleviate them.

2.1.3.1 Performance

The performance of a Web server is measured in terms of throughput, latency, execution time, and transaction time. Throughput represents the number of requests served in a given time period. Latency is the round-trip time between sending a request and receiving the response. Execution time is the time taken by a Web server to process its sequence of activities. Transaction time represents the period of time that passes while completing one complete transaction. Higher throughput, lower latency, lower execution and faster transaction times define high performing Web servers.

The performance of Web servers is limited by a number of factors such as the underlying network and network protocols, and the application itself. Therefore, per-

formance limitations are one of the fundamental drawbacks of Web servers. Most Web servers use a caching mechanism to reduce the execution time. Caching mechanisms have been detailed in [18] and [14]. A survey of caching mechanisms used in the Web are given in [19].

2.1.3.2 Reliability

Reliability is an overall measure of how well a Web server can maintain its service quality. The number of failures over a unit frame of time is a measure of reliability. Reliability is also a measure of in-order and assured message delivery. Web servers currently rely on protocols such as HTTP, which are inherently stateless and follow a best-effort delivery mechanism. It does not guarantee when the message will be delivered to the destination. This is one of the inherent limitations to an acceptable QoS. This can be somewhat resolved using new, reliable protocols, such as HTTPR and WS-Reliability [13], [1].

2.1.3.3 Integrity

Integrity is the degree to which a system or a component prevents unauthorized access or modification of data. Data integrity defines whether the transmitted data has been modified in transit. Data integrity is important for proper functioning and must be assured or it could corrupt a larger program and generate an error that is very difficult to trace. Web service transactions tend to be asynchronous and long running in nature. This makes it difficult to assure this QoS parameter.

2.1.3.4 Accessibility

Accessibility defines the capability of a Web server to serve a client's request. High accessibility of Web services can be achieved by building scalable systems [4]. Using many Web servers that are grouped together so that they act or are seen as one big Web server, is an effective technique (load balancing), that is utilized in many servers.

Load balancing spreads work between many servers in order to get optimal resource utilization and decrease computing time [3].

2.1.3.5 Availability

Availability determines if the Web server is ready for immediate use. There are a variety of issues which affect a Web server being available, namely:

1. Too much legitimate traffic
2. Distributed Denial of Service Attacks (DDoS) [5]
3. Large execution/transaction time of the server

Caching frequently-requested content is one of the ways to deal with excessive legitimate traffic [22]. An appropriately configured firewall can prevent DDoS attacks [12].

2.1.3.6 Interoperability

Interoperability is the ability of the Web server to work in a cross platform environment. Complying to standard specifications is important in achieving this goal.

2.1.3.7 Security

Since Web servers transfer data over the Internet, the data is exposed to many security threats. There are four aspects to providing security:

1. Confidentiality - A breach of confidentiality occurs when information that is considered to be confidential in nature has been, or may have been, accessed, used, copied, or disclosed to, or by, someone who was not authorized to have access to the information.
2. Integrity - Integrity means that data cannot be created, changed, or deleted without authorization.

3. Access Control - The prevention of unauthorized use of a resource, including the prevention of use of a resource in an unauthorized manner.

4. Authenticity - Authenticity means that the information is both genuine and original; the information is neither a fabrication nor a copy.

SSL is one of the important protocols used for securing remote access to a Web server [7], [9].

2.1.4 Web Server Trends

Web servers have come a long way since the first Web server, which was installed in 1991. Over the last decade, Web servers have gone through a major overhaul. The figures below are from a survey by Netcraft [15], which provides a good indication of Web server penetration now and in the last decade.

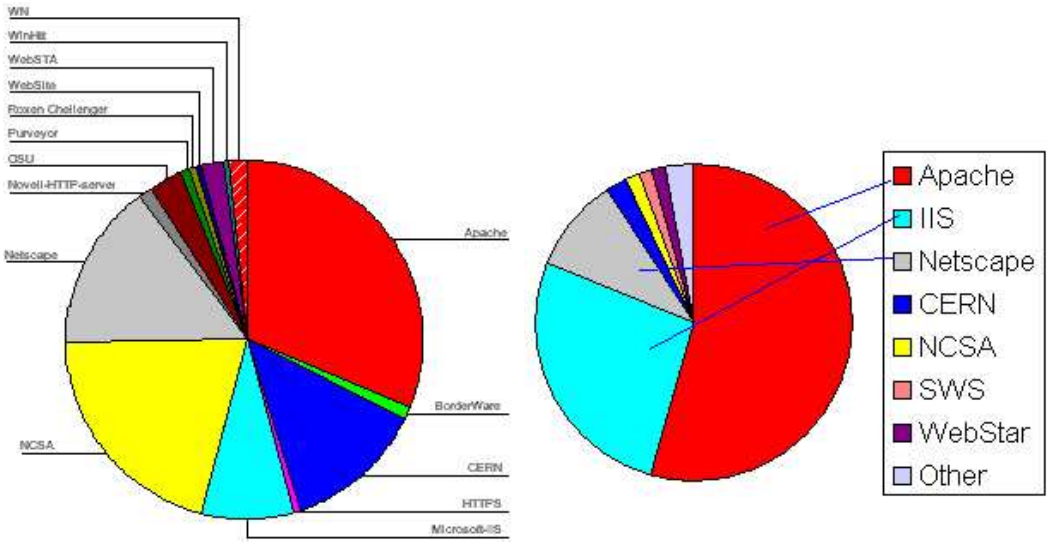

Figure 2.2. Comparison between Web server usage in 1997 and 2000

Courtesy: Brian Kelly (http://www.ariadne.ac.uk)

The most widely used Web server on the Internet today is Apache, which holds about 60% of the market share. The next most widely used server is the Microsoft

IIS, which holds about 30% of the market. Zeus and Sun capture the remaining market. The situation was very different just ten years ago, when there were numerous companies involved in the Web server business.

For some time now, Apache and IIS have commanded the largest share in the Web server market. Both are viable choices with their own sets of pros and cons.

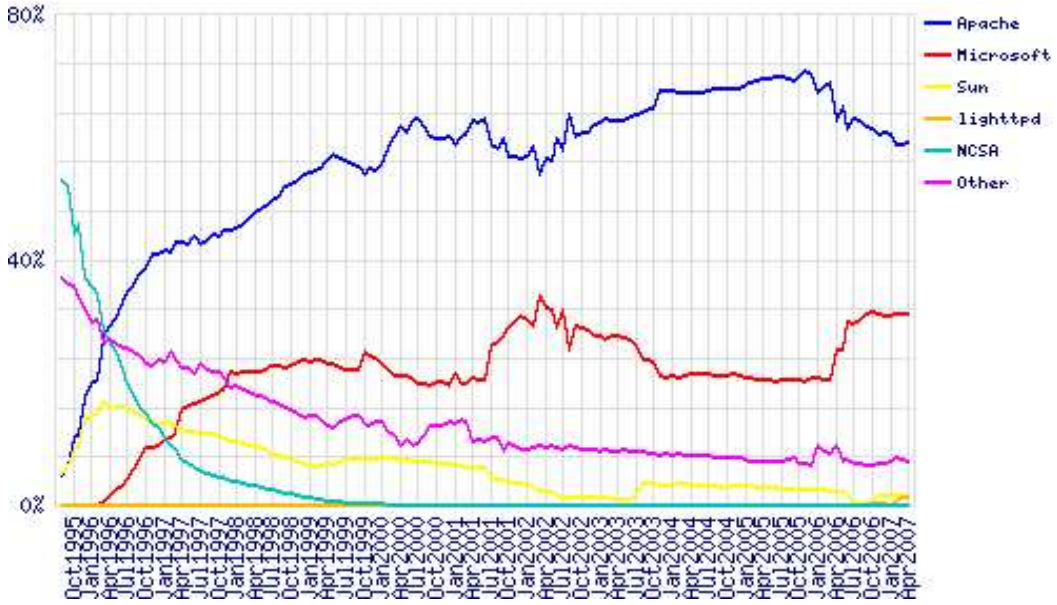

Figure 2.3. Market share for Web servers

Courtesy: Netcraft (http://www.netcraft.com)

2.1.5 Comparison between Apache and IIS

2.1.5.1 Execution Environment

IIS has been an optional component of the Windows Server operating systems since Windows NT 4.0. The current version at the time of writing is IIS 7.0 which is included in Windows Vista. It is not open-source and fairly expensive. It is designed and available to work only within the Windows environment. Although this limits the deployment platforms for IIS-based Web services, it also provides a number of benefits, including greater cooperation with the host operating system and easier management and control through a variety of standard OS tools and utilities.

Apache, on the other hand, is free and open-source and can be installed on an OS such a Linux, which is also free. The current 2.x versions are supported under a wide range of mainstream operating systems, including Windows, Linux, Unix and Mac OS X, and some non-mainstream OSs such as VMS.

Both provide good multithread and multimodule support.

2.1.5.2 Dynamic Components

The primary dynamic environment for development within IIS is Active Server Pages (ASP). This is a generic term for a solution that allows code to be embedded into HTML pages. These ASP pages are parsed by the server before being supplied to the client as HTML. The ASP system allows developers to work in a number of different languages, including Visual Basic, VBScript, JavaScript, Java, and C/C++, along with other open source alternatives, such as Perl and Python. In addition, IIS also supports CGI methods along with its own suite of filtering and execution systems in the form of ISAPI filters.

Apache is designed to work with a wide range of languages, through the CGI model, or through the use of dynamic modules by directly incorporating the language interpreter into the Apache environment. This significantly speeds up the execution of dynamic components for languages like PHP, Perl, and Python. ASP can be supported under Unix using the Apache mono module. However, the Microsoft.NET environment cannot be emulated in Unix.

2.1.5.3 Security and Authentication

IIS benefits from close integration with the operating system. Since the same services that provide authentication and security for the main system also power the IIS, it reduces management overhead.

Apache's security and administration system is not as well-integrated with the OSs it supports. However, there are modules and directories that support a variety of different authentication and security sources.

In terms of secure transactions, both systems support the SSL encryption technology, and can be used with IPSec implementations.

2.1.5.4 Management

IIS offers a number of different interfaces to modify the configuration of the system. Although the underlying configuration is primarily stored in an XML-based text file, the IIS system enables the admin to change the configuration and the underlying file while the system is still running. Because the file is XML, it has a more rigid structure, yet is still flexible enough to support the different configuration options. There is also a GUI-based interface for editing and configuring different components. The XML format also makes it easy to export and import configuration information between machines to share configuration details.

The only method of administering Apache is through a simple text-based configuration file. Although a variety of command line, Web, and GUI interface tools are available, it relies on the central text file to update the configuration. Using a text file this way has some advantages. It is easy to share configuration information between machines by copying the relevant content from the text file. The primary disadvantage is that it is relatively easy to corrupt the file and therefore upset the configuration. The latest version of Apache allows for soft restart (i.e. changing configurations without having to restart the server).

2.2 Proxy servers

There are two major performance problems when accessing Web services:

1. Total Service Time - It is defined as the elapsed time between a client sending a request and receiving the required page. Increase in this time leads to high frequencies of timeouts and increase in access times for the client.

2. Congestion - High loads on networks and Web servers lead to long delays and increased CPU and/or memory utilization.

Proxy servers with caching techniques help in alleviating these problems.

2.2.1 Definition

A proxy server is a server which services client requests by forwarding them to other servers, or optionally serves the request itself. A client first connects to the proxy server and requests some resource, such as a file or an image. The proxy server then provides the resource by connecting to some specified servers and requesting the resource on behalf of the client. It can optionally serve the request itself and/or alter the client's request or the server response.

A proxy server can function as a cache of frequently requested resources, serve as a filter of content, anonymize client requests and provide transparent redirections. Squid is an example of a popular proxy server [17].

Proxy servers can be applied in two ways:

1. Client-side - The pages requested by the client are copied to the local PC or other proxy servers. This reduces the access time for subsequent requests. However, this technique is not transparent, and needs to be specially implemented at the client.

2. Server-side - Aims to relieve Web server overload and reduce access times.

2.2.1.1 Server-side Proxies

A server-side proxy (also known as a reverse proxy) is a proxy server that is installed in the neighborhood of one or more servers. It is implemented in front of

Web servers. Therefore, it differs from the usual client proxy that is configured in the client's browser. All connections coming from the Internet addressed to the main server are routed through the proxy, which may either deal with the request itself or pass the request (modified or unmodified) to the main Web server.

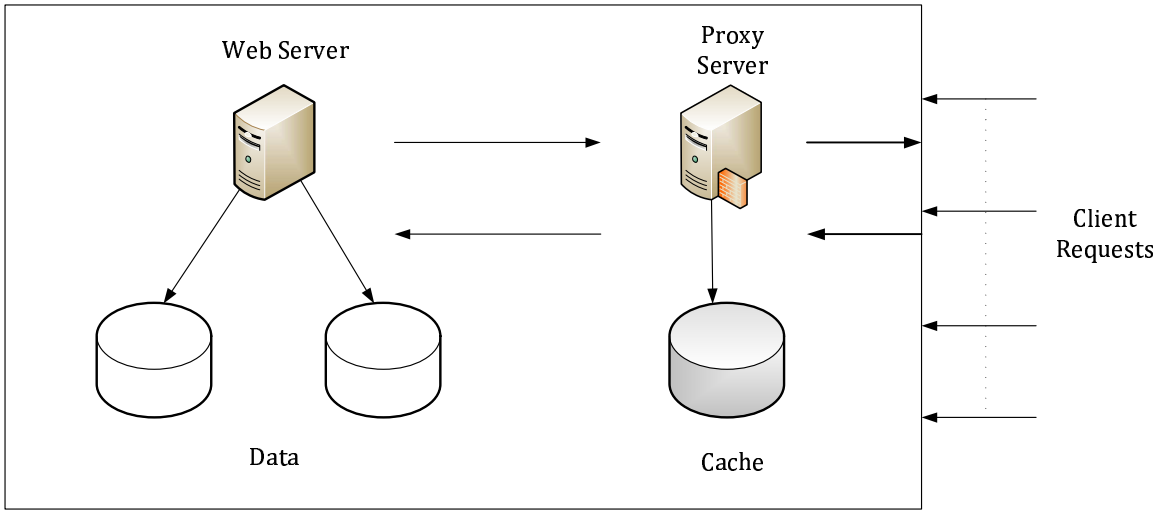

Figure 2.4. Reverse Proxy

There are several reasons for installing reverse proxy servers:

1. Security - Provides an additional layer of defense.

2. Encryption - Can be utilized for SSL encryption in secure websites, if equipped with SSL acceleration hardware or software.

3. Load Distribution - Distributes load to several servers.

4. Static Content Cache - Can offload a large amount of static data from main Web servers. Reverse proxies that serve as caches can satisfy a large number of client requests, thus reducing the load on the main Web servers.

2.2.2 Performance Parameters

There are a number of parameters on which the performance of a proxy depends:

1. Hit Rate - Hit Rate measures the percentage of all accesses that are satisfied by data in the cache. A high hit rate signifies fewer accesses from the disk and a reduced access time.

2. Byte Hit Ratio - Byte Hit Ratio measures the percentage of the number of bytes that hit in the cache as a percentage of the total number of bytes requested. This is an important factor if the objects requested are relatively large.

3. Cache Size - It is important to configure an optimum cache size. If it is set too small, there will be too many disk accesses resulting in high latency. On the other hand, if it is set too large, it will consume much more memory than it needs. This results in poor system performance because of heavy load. If the system happens to run out of physical memory, the OS will swap the application out of memory, which further deteriorates performance.

4. Replacement policy - As proxies have finite storage capacity, it eventually becomes necessary to evict one or more cached objects to make place for a new object. The heuristic that it uses to choose the entry to evict is called the replacement policy. A good replacement policy is required to select a cached object for removal which will least affect the performance of the cache. Commonly used policies include:

    - LRU (and variations) - Evicts Least Recently Used among objects present in cache. It exploits "locality of reference", where few resources are requested repeatedly by clients. It is simple and robust to implement, but does not take into account different object sizes.
    - LFU - Evicts Least Frequently Used among objects present in cache. This policy works well in situations when the cache is suddenly flooded by documents that are referenced only once. The objects which have been highly requested are retained and the least used are evicted.

- LRV - Takes into account size, recency and frequency of objects and evicts the object with the Least Relative Value. However, it suffers from high overload in implementation.
- Greedy Dual Size - Combines recency of reference, object size and retrieval cost. Based on the object's size and cost, an initial value is assigned to it when it enters the cache. This value decreases if there are no references to the object. The object with the least value is evicted.

5. Maximum Object Size - A large object size will result in a high byte hit ratio, while a small object will result in faster speed.

CHAPTER 3

APPLICATION-LEVEL OFFLOADING

Offloading applications or parts of applications for processing on an separate system or device leaves the main CPU free to accept more requests. Many kinds of offloading have been explored. TCP/IP offloading is one of the most commonly used approaches, where the networking part of applications is offloaded onto a device which has specialized hardware to speedup TCP/IP processing. It is detailed in the section below.

3.1 Related Work

3.1.1 TCP Offload Engine

The Transmission Control Protocol (TCP) is one of the core protocols of the Internet. It is used for establishing connections between networked hosts, so that they can exchange data over stream sockets. The protocol guarantees reliable and in-order delivery of data from sender to receiver. TCP also distinguishes data for multiple connections by concurrent applications running on the same host.

Since TCP is a connection-oriented protocol, it has several complex features that have a considerable protocol overhead.

1. Connection establishment is via a three-way handshake. A number of messages pass between the end points before data flow actually begins.

2. Since it is a reliable protocol, acknowledgments are also sent. This adds to the protocol load.

3. In-order and reliable delivery uses checksum and sequence numbers, both of which require processing.

4. Congestion and flow control is also provided by means of a sliding window.

TCP Offload Engine or TOE is a technology used in network interface cards. A TOE is a specialized network device that implements a significant portion of the TCP/IP protocol in hardware, thereby offloading TCP/IP processing from software running on a general-purpose CPU. The technology aims to reduce the load on the server CPU by shifting TCP/IP processing tasks to the network device. This leaves the CPU free to run its applications, so users get their data faster. It is primarily used in conjunction with high-speed network interfaces, such as 100 and 10 gigabit Ethernet, where processing overhead of the network stack becomes significant [6].

TOEs have been proposed as an answer to the increasing demand put on servers by gigabit connections. In data transfers, the system CPU may need to repeatedly interrupt applications to request data from the disk. This I/O and memory processing slows down the actual applications. Therefore, even servers running at high speeds (greater than 1 GHz) have trouble keeping up with high data rates.

There are many research publications on the benefits of TCP offloading. The impact of TCP offloading on Web servers was first studied in [11], where a detailed modeling of TCP offload was presented and the SPECweb99 Web server benchmark was used in order to assess the impact of offloading on the overall server performance. Results indicated that a carefully designed offload engine can almost double the performance. A system with a dedicated network interface card for TCP offloading on an SMP processor was studied in [21], and realized a performance benefit of 600% - 900%.

TOE offers performance gains specifically for Web servers and server applications which typically have to support a large and changing number of simultaneous connec-

tions, especially where the HTTP protocol requires a new TCP connection for every object.

3.2 Research Goals

Network devices have become increasingly sophisticated in the past decade. From packet processing (routers and switches) to complicated QoS and security applications (cryptoprocessors), the advances made in the field of networking devices is appreciable. Judging by the processing capabilities of networking devices today, though they have not reached the level of general purpose processors, they can deal with a high level of complexity and processing power.

In this thesis, I extend the concept of TCP offloading to a more general problem: Given the advancements in networking devices, will offloading entire applications onto a network device provide a performance enhancement if used in conjunction with a Web server? Network device technologies indicate that if there is a proper partitioning of tasks between the host processor and the network device, considerable performance improvements can be expected. Inclusion of dedicated cryptoprocessing engines on many network devices provides a basis to expect that security-related workloads, such as SSL termination have a considerable chance to improve the overall server performance.

I proceed in the following way to prove this hypothesis:

1. **Step A** - Measure performance of the Web server using standard benchmark workloads without using any network device. This is known as an **IA-only system** where IA stands for Intel Architecture. Data transfer occurs directly between the server and client.

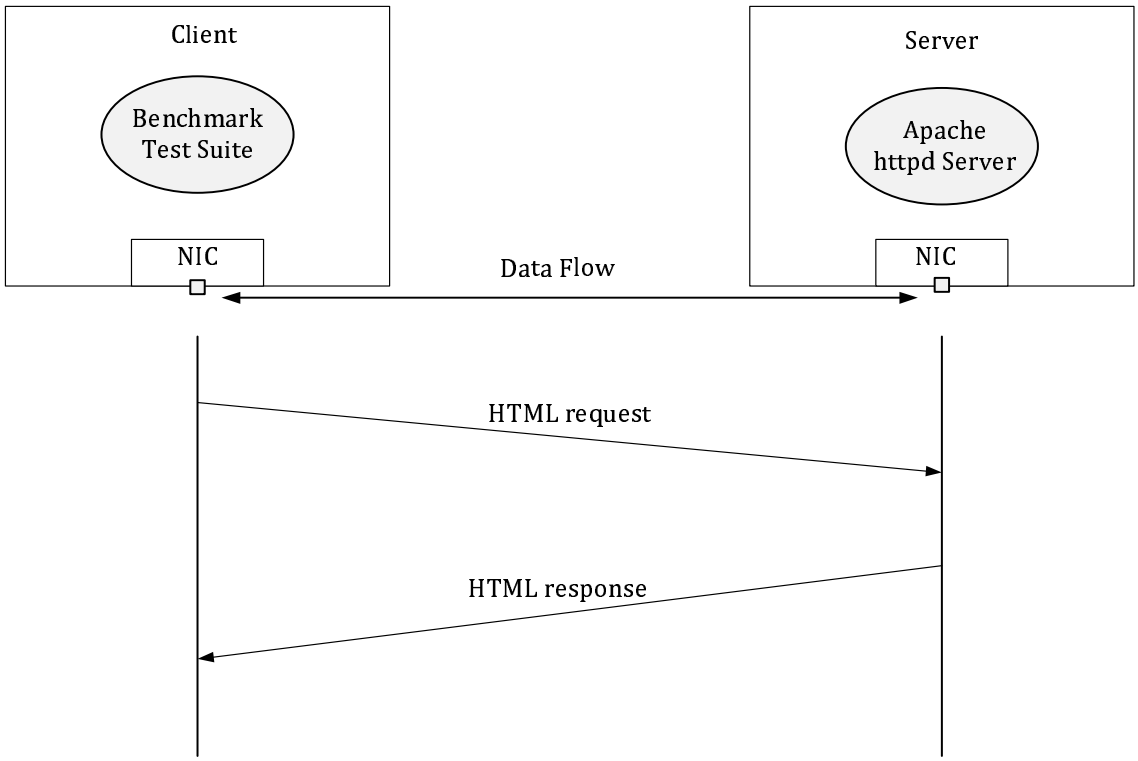

Figure 3.1. Step A: IA-only system

2. **Step B** - Measure the performance of the Web server using standard benchmark workloads and a proxy server configured in the reverse mode on the main system.

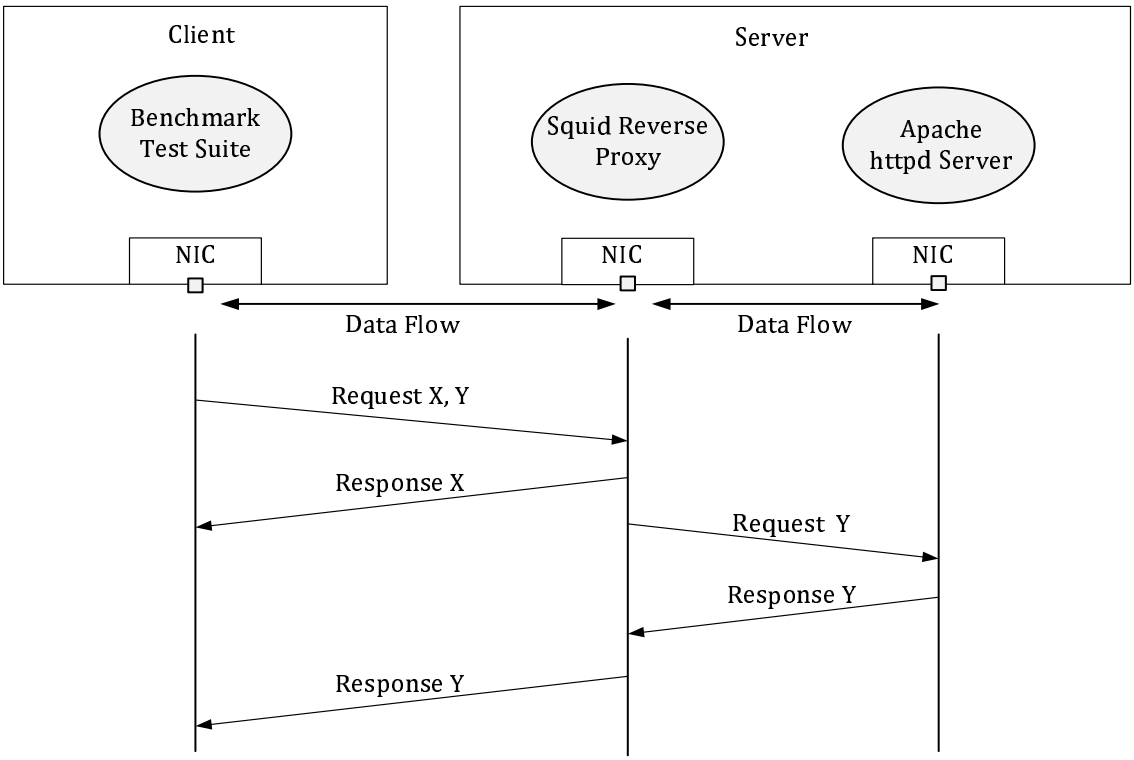

Figure 3.2. Step B: IA system - Using a reverse proxy server to transfer data between the server and client

3. **Step C** - Extend the concept of implementing proxy servers to the network processor. Implement a proxy on the network device which serves some of the client requests and forwards the rest to the main server. This is known as an **IA/IXA system**, where IXA stands for Intel Extended Architecture.

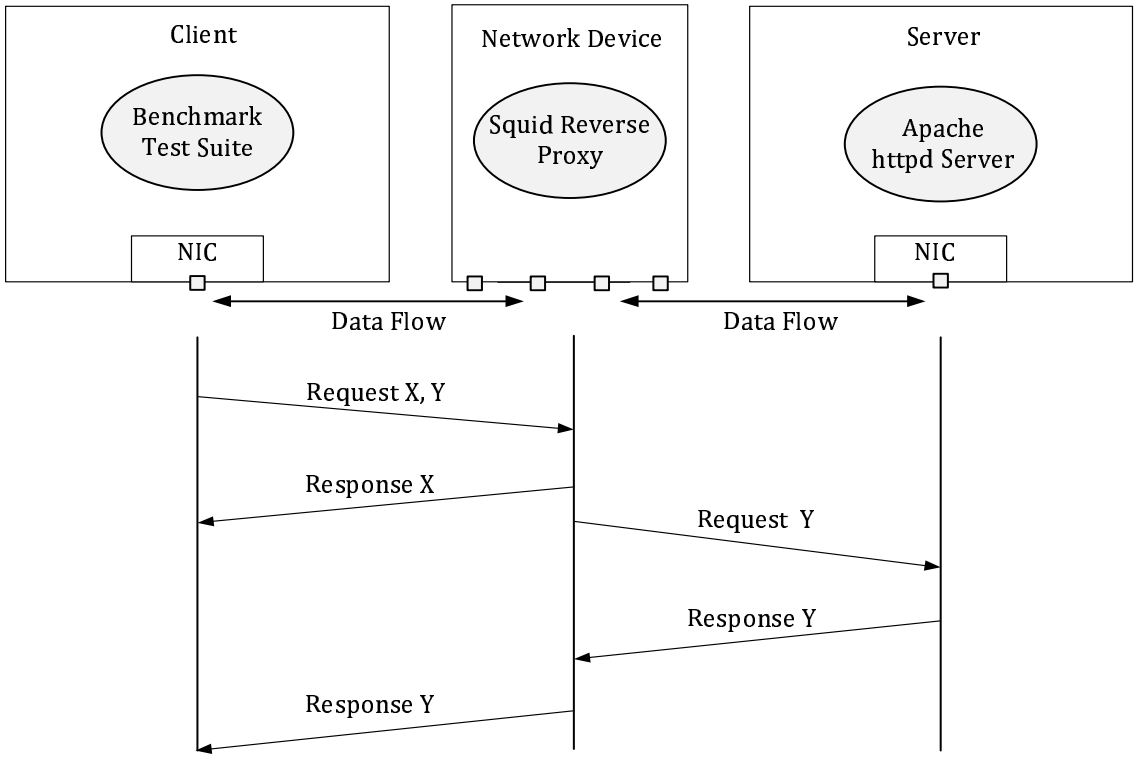

Figure 3.3. Step C: IA/IXA system - Using the network device to offload a set of applications

Steps A, B and C provide a systematic comparison between an IA-only system, an IA system employing a proxy server to speedup performance and an IA/IXA system where the proxy server is implemented on the network processor.

It is possible to intuitively decide which applications perform better on the network device. A good example of such an application would be any static workload which allows caching (HTTP), and is not very processor hungry. Any network device that allows for caching techniques to be employed, can respond to a request that can

directly be served from its cache. Since network caches are generally smaller and on-chip, considerable increase in performance can be expected.

Another application could be secure workloads. The cryptographic algorithms in use today are usually extremely complex, in terms of key size, key exchange, encryption/decryption and so on. The SSL protocol, for example, is very computationally intensive. It uses a public key infrastructure with key sizes of 128 bits and more. Hence, there is significant performance loss. Most network devices have their own cryptoprocessing engines. Therefore, by offloading security applications, performance gain can be expected.

3.3 Design and Component Considerations

3.4 Network Processors

3.4.1 Motivation

All components on a network usually perform some form of packet processing. Previously, data rates were slow and networking protocols fairly simple. Therefore, packet processing in the network components was straightforward and mostly implemented in software running on general-purpose processors. Today, core network speeds approaching 40Gbps and edge networks operating at 2.5Gbps are starting to emerge. Current networking trends reflect the introduction of more and more complex protocols to support the demands of security, differential QoS and the shift to IPv6. These applications have requirements that are far beyond the developments in bus and clock latencies in general-purpose processors, and also need significant processing power.

The demand for high bandwidth networks has driven the evolution of network equipment design. The initial designs used only CPUs. However, many of the designs were not useful for network programming. Furthermore, little use was made of the more complicated components such as floating point units. Since the demand

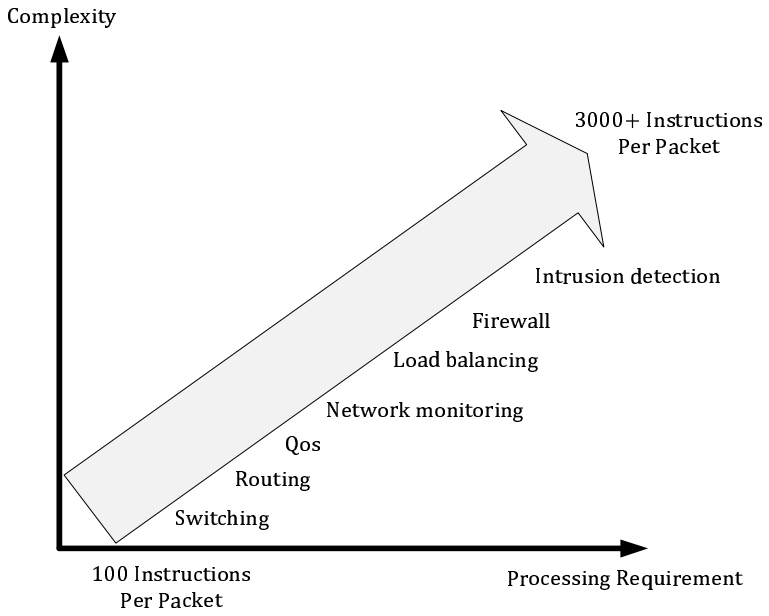

Figure 3.4. Increased Complexity of Processing

for bandwidth is increasing faster than CPU speeds, network equipment design began to shift away from general purpose processors to processors designed for specific applications.

Hardware-based solutions using ASICs (Application-Specific Integrated Circuit) are customized for a particular use. Well-designed ASICs are designed to be much faster than conventional CPUs, however they have a downside of being difficult and expensive to develop. Moreover, they have limited programmability. Moreover, as networks have developed, it has become apparent that network devices need to be flexible so as to support new protocols that are developed alongside. This calls for a level of re-programmability in ASICs that they cannot provide.

Network processors are programmable chips like general purpose microprocessors, but are optimized for the packet processing required in network devices. They have helped bridge the gap between ASICs and CPUs, by being as programmable as CPUs but as fast as ASICs.

3.4.2 Network Processor Architectures

There are a few architecture features that are common to most network processors:

1. **Multiple core and multiple threaded core systems**

   Network processors are software programmable, similar to general purpose processors. Software is not suitable for basic network applications, since packet processing is greatly slowed down, as has been indicated above. The major difference in NPs is that they employ multiple programmable processing engines (PPEs) or processing cores within a chip. They are known by different names - microengines (Intel), channel processors (C-5) or task optimized processors (Lucent).

   The functionality within the PPE is manufacture dependent. Many are based on Reduced Instruction Set Computer (RISC) cores. Bit manipulation is a critical feature for network processors and therefore is an added functionality to the RISC core. Since RISC architectures deal with very simple instruction sets, more complicated operations require multiple instructions. Therefore, RISC-based network processors have usually have their PPEs arranged in parallel. An alternative style is to organize them in a pipeline form, or a combination of both pipeline and parallel styles.

   Most network processors also employ some form of multithreading on PPEs to maximize their performance. During a typical execution, a PPE will be idle if it needs a resource that is currently in use. Increased idle time has an adverse effect on the overall system utilization. Therefore, multithreading is used, where multiple tasks can be run on each PPE, and the PPE can switch between tasks. Hardware support is usually provided to ensure fast switching between tasks. This is another advantage of network processors over general purpose processors, which require several instruction cycles to switch tasks. Software

switching usually means that PPEs are likely to remain idle, especially if the switching requires an external memory access.

The downside to multicore/multithreaded PPEs is the increased complexity in coding.

2. **Dedicated hardware for common networking operations**

   Network processors provide specialized hardware or integrated coprocessors for commonly performed networking tasks. Typical functionality includes encryption/decryption, lookup engines, queue management and CRC calculation. Other functionality targeted for specific applications could also be added.

3. **High speed memory interfaces**

   Network processors require memory for packet and queue information, program code, lookup tables etc. Each PPE has a small amount of internal memory, that is local to it. In addition, many network processors also have a shared area of memory, usually used for storing lookup tables and packet information. However, the amount of available internal memory is limited, and therefore interfaces to external memory are also provided.

   Most network processors usually have at least two external memory interfaces - one to Static Random Access Memory (SRAM) and one to Dynamic Random Access Memory (DRAM). Packet data typically requires a large buffer storage, so in general it is stored in DRAM. Queuing information is usually stored in SRAM.

4. **Switching fabric interface**

   Many of the network processors are designed for routing applications and provide an interface to a switching fabric.

5. **Control Processing**

The goal of a Network Processor is to process packet data at wire speed. However, certain kinds of packets, such as management and control packets need not to processed at wire speed. They have much more complex processing requirements, which would in turn slow down the packet throughput. Therefore, they are separated from the main path and passed onto a separate processor known as a control processor or an **XScale**. Network processors usually contain an integrated core for control processing.

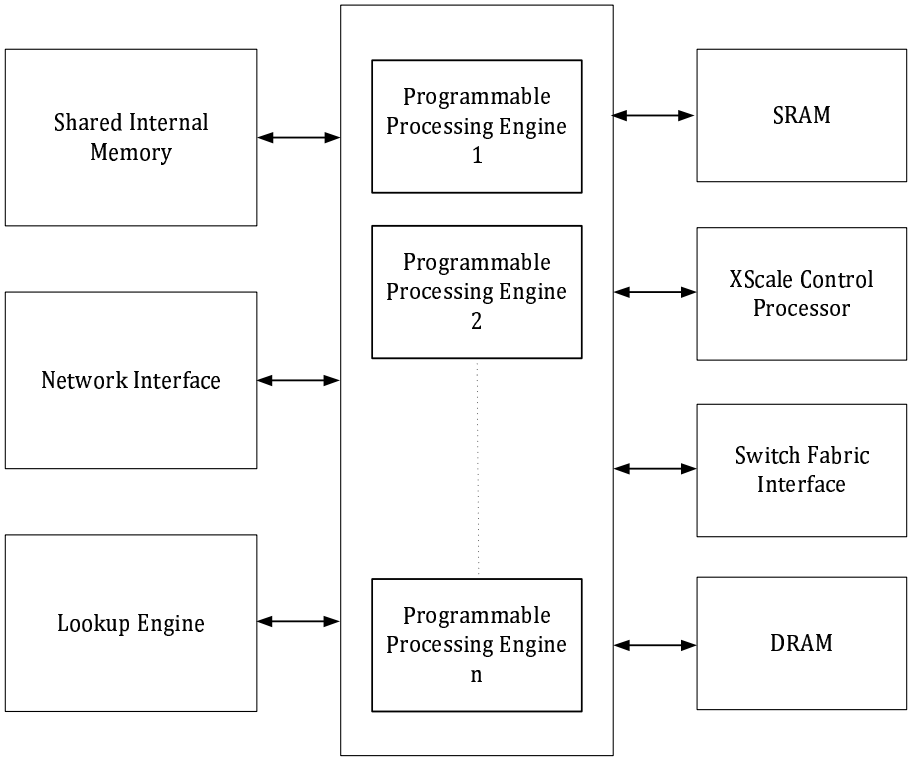

Figure 3.5. Generic Architecture of a Network Processor

3.4.3 Applications

A few of Network Processor applications are listed below:

1. **Smart switches**

   The tremendous increase in Internet traffic overload Web servers with a high volume of requests. Many ISPs resort to a cluster-based approach to provide

a cost-effective, scalable and reliable solution to the problem. Any distributed system has a problem of partitioning tasks between components. To make this distribution of tasks transparent to end users, a switching device is placed as a common interface in front of the server cluster. They operate on layer-5 information (request content or application information) and are termed as content-aware switches.

ASIC based switches have little flexibility in terms of re-programmability, although they have high processing capabilities. On the other hand, switches based on general purpose processors do not provide a satisfactory performance because of interrupt issues, increased stack overhead and so on.

The above problems can be solved by using network processors for content-aware switching [24]. They do not suffer software hindrances such as stack overhead and are also programmable to an extent. Moreover, their architecture and instruction set are optimized to provide a good throughput for packet processing (using multiprocessing, multithreading etc.).

2. **Web server accelerators**

   Web servers are limited by a number of factors, such as the underlying operating system, interrupt processing etc. One technique to improve performance of Web servers is to cache frequently requested content, so that there is less overhead while requesting these pages. Such caches are termed as httpd or Web server accelerators.

   The local interconnect within a server, such as the PCI bus remains a performance bottleneck, because all data transferred over the network is sent over this interconnect. This bottleneck can be reduced by caching data directly on a programmable network interface, such as a network processor. The cache can

reside on DRAM or SRAM, and the network processor can store and access the data within the cache easily and quickly.

3. **SSL Offloading devices**

   Security is given high importance in the Internet, even if it is at the cost of a lower performance. The commonly used SSL protocol is very computationally intensive. Traditional designs have addressed network security by adding a co-processor. As data rates increase, co-processors have practical limitations. Integrating security functions within a network processor makes it possible to encrypt and decrypt network traffic at high speeds.

I use the NFE-i8000, containing the IXP2855 network processor as the network device for offloading, in conjunction with the Apache httpd Web server and Squid functioning as a reverse proxy server. The motivation behind this design is elaborated in the following section.

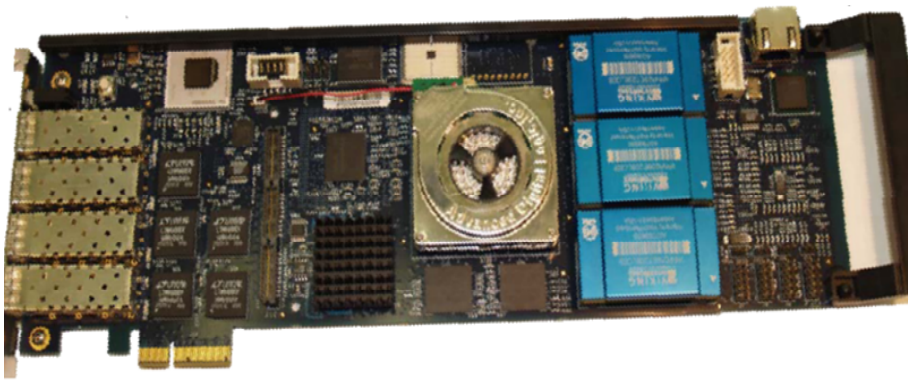

Figure 3.6. NFE-i8000

Courtesy: Netronome Inc.

3.5 Web servers

3.5.1 Apache Web server

For my thesis, I have chosen to use Apache httpd Web server for the following reasons:

1. Most widely used - According to the survey by Netcraft on Web server penetration [15], the Apache Web server holds the largest market share and accounts for over 60% of the Web servers deployed throughout the Internet, ranging from personal to enterprise servers. Therefore, it follows that positive results on performance evaluation using the Apache server will have a considerable impact.

2. Open source - The Apache Web server is freely and easily available. The Web server code can also be modified according to convenience.

3. Modular design - The modularized software design of the Apache Web server saves on memory and performance by loading only the required functionalities.

3.6 Proxy Server

3.6.1 Squid

The proxy server that I have chosen for my thesis is Squid [17] deployed in the reverse proxy mode. Squid is a widely used proxy server and web cache daemon. It can also serve as an SSL terminator. Squid uses the Internet Cache Protocol (ICP) [20] for coordinating Web caches. Reasons for choosing Squid as the reverse proxy include:

1. Widely used - Squid is a widely used proxy server, and very popular on Unix machines.

2. Open source - Squid is free and easily available. Squid can also be cross-compiled for different architectures such as ARM (which is the architecture of the XScale processor on the NP).

CHAPTER 4

PERFORMANCE EVALUATION

4.1 Benchmarking

A benchmark is a way to measure system performance. The idea behind all benchmarks is essentially the same: A process that is typical for a system is performed and various parameters which are useful to evaluate the performance of that system are measured under standard configurations. This test is then repeated for different workloads and for different system configurations.

A typical approach for benchmarking a Web server is to simulate a large number of clients, and then request a set of pages of varying lengths, so that results can be obtained for both large and small files.

4.1.1 SPECweb2005

For performance measurement, I use SPECweb2005, which is a software benchmark product developed by the Standard Performance Evaluation Corporation (SPEC). It measures a Web server's ability to handle both static and dynamic page content. It also provides the capability of measuring both SSL and non-SSL requests. There are three different workloads provided for this benchmark:

1. Banking - Designed to simulate an online banking system. Banking provides an HTTPS-only workload.

2. ECommerce - Designed to simulate a Web server that sells computer systems. This includes allowing end users to search, browse, customize, and purchase products. ECommerce provides a mix of HTTPS and HTTP workload.

3. Support - Designed to simulate a vendor's support website. Users can search for products, browse a listing of available products, filter a listing of available downloads based upon certain criteria, and then download files. Support provides an HTTP-only workload.

The benchmark clients run the application program (written in java) that sends HTTP requests to the server and receives HTTP responses.

4.1.1.1 Design

SPECweb2005 has four major logical components: the clients, the prime client, the Web server, and the back-end simulator (BeSim):

1. Client - The benchmark clients run the application program that sends HTTP requests to the server and receives HTTP responses.

2. Prime Client - The prime client initializes and controls the behavior of the clients, runs initialization routines against the Web server and BeSim, and collects and stores the results of the benchmark tests. It can run on the same physical system as one of the clients.

3. Web Server - The Web server is a collection of hardware and software that handles the requests issued by the clients.

4. Back-End Simulator (BeSim)- BeSim is intended to emulate a back-end application server that the Web server must communicate with in order to retrieve specific information needed to complete an HTTP response (customer data, for example).

4.1.1.2 Implementation

The SPECweb2005 benchmark is used to measure the performance of HTTP servers. The HTTP server workload is driven by one or more client systems, and

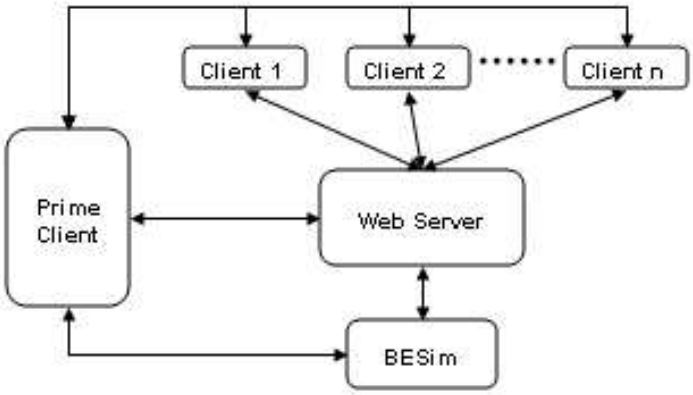

Figure 4.1. SPECweb2005 Logical Components

Courtesy: SPEC

controlled by the prime client. Each client sends HTTP requests to the server and validates the server responses. When all of the HTTP requests have been sent and received, the QOS criteria met for that web page transaction is recorded by the client.

Prior to the start of the benchmark, one or more client processes is started on each of the client systems. These processes either listen on the default port (1099) or on another port that can be specified in the configuration files. Once all client processes have been started, the client systems are ready for workload and run-specific initialization by the prime client.

The prime client will read in the key value pairs from the configuration files. Upon successful completion, it will initialize each client process, passing each client process the configuration information read from the configuration files, as well as any configuration information the prime client calculated (number of load generating threads, for example). When all initialization has completed successfully, the prime client will start the benchmark run.

At the end of the benchmark run, the prime client collects result data from all clients, aggregates this data, and writes this information to a results file. When all

three iterations have finished, an ASCII text report file and an HTML report file are also generated.

The prime client controls the phases of the benchmark run. These phases are illustrated in the diagram below:

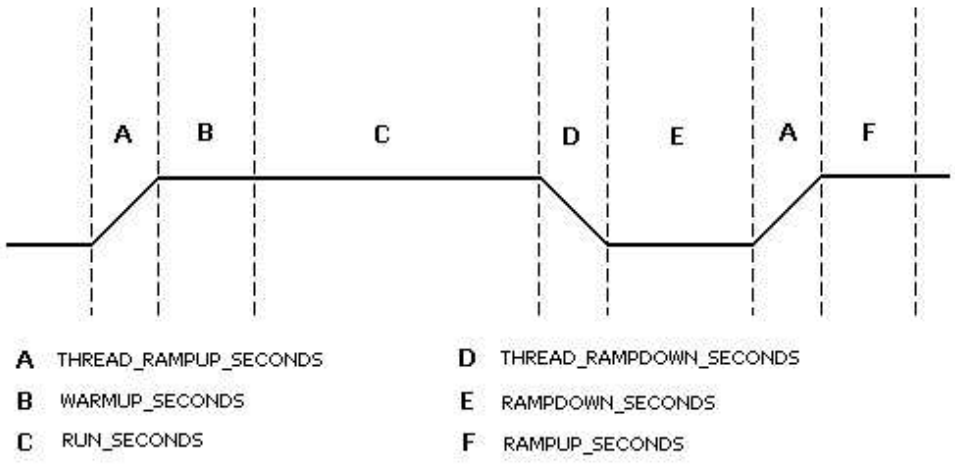

Figure 4.2. Benchmark Phases

Courtesy: SPEC

- The thread ramp-up period A - The time period across which the load generating threads are started. This phase is designed to ramp up user activity.
- The warm-up period B - The time during which the system can prime its cache prior to the actual measurement interval. At the end of the warm-up period, all results are cleared from the load generator, and recording starts anew.
- The run period C - The interval during which benchmark results are recorded.
- The thread ramp-down period D - It is the period during which all load-generating threads are stopped, and is the inverse of A. Although load generating threads still make requests to the server during this interval, results are not recorded.

- The ramp-down period E - The time given to the client and server to return to their normal states. This is primarily intended to assure sufficient time for TCP connection clean-up before the start of the next test iteration.

- The ramp-up period F - It replaces the warm-up period, B, for the second and third benchmark run iterations. It is presumed at this point that the server's cache is already primed, therefore, this time period is quite short.

4.1.1.3 Workload Generation and Analysis

SPECweb2005 follows a page based model. Each page is initiated by a dynamic GET or POST request, which runs a dynamic script on the server and returns a dynamically created webpage. Associated with each dynamic page, are a set of static files or images, which the client requests right after the receipt of the dynamically created page. The page returned is marked as complete when all the associated images/static files for that page are fully received.

The workload generated is based on page requests, transition between pages and the static images accessed within each page. The QoS requirements for each workload are defined in terms of two parameters, Time_Good and Time_Tolerable. QoS requirements are page based and Time_Good and Time_Tolerable values are defined separately for each workload. For each page, 95% of the page requests (including all the embedded files within that page) are expected to be returned within Time_Good and 99% of the requests within Time_Tolerable. Very large static files (i.e. Support downloads) use specific byte rates as their QoS requirements.

The validation requirement for each workload is such that less than 1% of requests for any given page and less than 0.5% of the all page requests in a given test iteration fail validation. Small session workloads usually have high validation errors because of insufficient requests to obtain steady state readings (at least 100 requests).

The following tables represent the workload page mix percentage:

Support	Mix%
catalog	11.71%
download	6.76%
file	13.51%
file catalog	22.52%
home	8.11%
product	24.78%
search	12.61%

Table 4.1. Support Workload

ECommerce	Mix%
billing	3.37%
browse	11.75%
browse product	10.03%
cart	5.30%
confirm	2.53%
customize1	16.93%
customize2	8.95%
customize3	6.16%
index	13.08%
login	3.78%
product detail	8.02%
search	6.55%
shipping	3.55%

Table 4.2. ECommerce Workload

Banking	Mix%
summary	15.11%
add payee	1.12%
bill pay	13.89%
bill pay status	2.23%
check detail html	8.45%
check image	16.89%
change profile	1.22%
login	21.53%
logout	6.16%
payee info	0.80%
post check order	0.88%
post fund transfer	1.24%
post profile	0.88%
quick pay	6.67%
request checks	1.22%
req xfer form	1.71%

Table 4.3. Banking Workload

For each workload, there are two types of files:

1. Non-scaling filesets that do not grow with the load.

2. Scaling filesets that grow linearly with the number of simultaneous connections.

4.1.1.4 Key Java Classes

The three classes invoked are specweb, specwebclient, and reporter. "specwebclient" is invoked on one or more client systems to start the client processes that will generate load against the HTTP server. Once invoked, specwebclient listens on the assigned port waiting for the prime client's instructions.

The prime client is started by invoking specweb on the prime client system. "specweb" reads in the relevant configuration files and then lets the SPECwebControl class handle benchmark execution. SPECwebControl then creates a RemoteLoadGenerator to handle communication between the prime client and the client processes. RemoteLoadGenerator communicates with the specwebclient processes via Java's Remote Method Invocation (RMI). The RMI methods called by RemoteLoadGenerator to control specwebclient are:

- createThreads(): Creates the number of load-generating threads/processes that will send requests to the server.

- setup(): Handles initialization of static workload class variables.

- start(): Starts each load-generating thread. start() corresponds to the beginning of Phase A (thread ramp-up).

- stop(): Stops each load-generating thread. stop() corresponds to the beginning of Phase D (thread ramp-down).

- getHeartbeat(): Tells the prime client that this client process is still alive.

- waitComplete(): Gives the load-generating threads a finite amount of time to clean up, before attempting to terminate all threads. It then displays the benchmark run statistics for that client process.

- exit(): Allows the prime client to kill the client process. This is used at the end of a complete benchmark run.

- getStatistics(): Returns the results collected by this client process during the benchmark run. This is called at the end of each iteration by the prime client.

- clearStatistics(): clears all results collected prior to the beginning of Phase C (run period).

- isReady(): Returns true when all load-generating threads have been created on the client.

- cleanUp()(Optional): Gets rid of objects created during the run that will be recreated in any subsequent run, including the LoadGenerator object on the client. It is optional because Java has a garbage collector which performs the same task.

The diagram below illustrates the various methods.

The reporter class is invoked only to create the ASCII and HTML results.

4.1.1.5 Performance Measurement

The SPECweb2005 individual workload metrics represent the actual number of simultaneous sessions that a server can support while meeting quality of service (QoS) and validation requirements for the given workload. In the benchmark run, a number of simultaneous sessions are requested. These sessions correspond to the number of load-generating processes/threads that will continuously send requests to the HTTP server during the benchmark run. Each of these threads will start a user session that

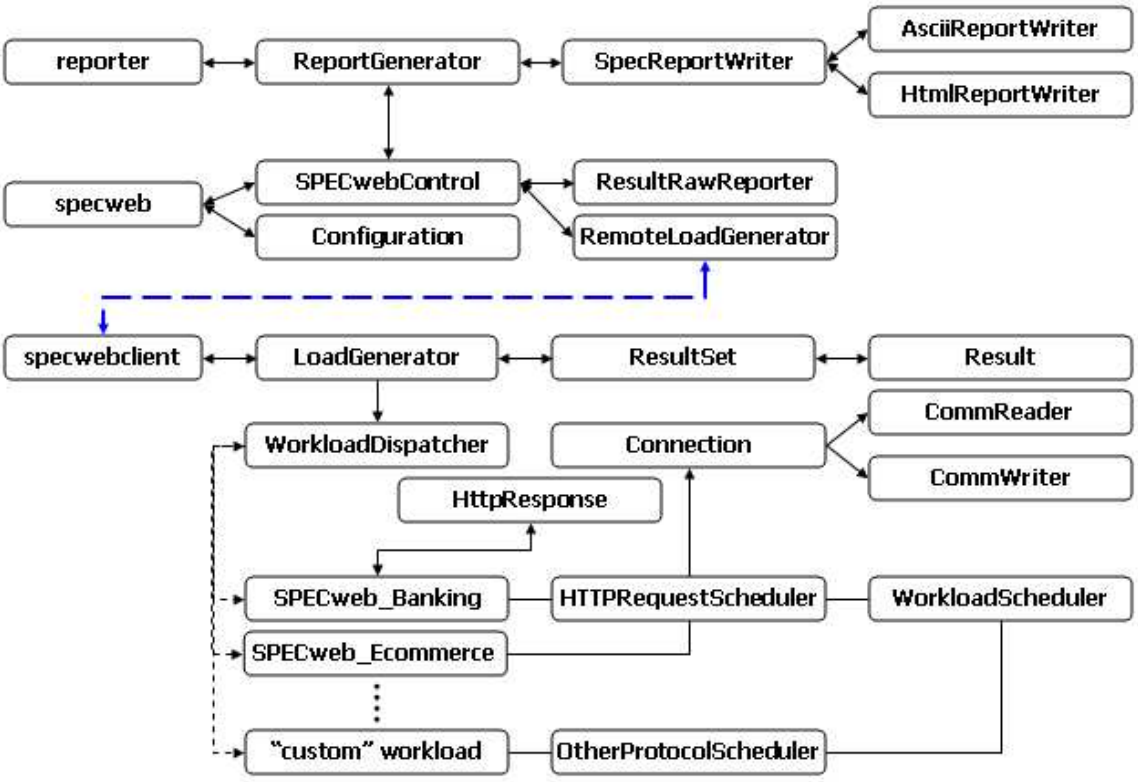

Figure 4.3. Key Java Classes in SPECweb2005

Courtesy: SPEC

will traverse a series of workload-dependent states. Typically, each user session would start with a single thread requesting a dynamically created file or page. Once the user session ends, the thread will start a new user session and repeat this process. This process is intended to represent users entering a site, making a series of HTTP requests of the server, and then leaving the site.

4.2 Measurement Setup

4.2.1 System Configurations

There are three system configurations for which measurements have been taken:

1. IA-only setup: Data flow is only between the client and server systems. There is no network processor involved in the communication. This is further divided into the following scenarios:

   - Web server only - All requests from the client are handled only by the Web server.
   - Web server and proxy - All requests from the client are first forwarded to the proxy. If the requests cannot be serviced by the proxy, they are then sent to the Web server.

2. IA/IXA setup: Data flow between the client and server goes via the network processor, which acts as the proxy server. All requests from the client are first sent to the proxy on the NP. If the proxy cannot service the request, it forwards it to the main Web server.

4.2.2 System Parameters

4.2.2.1 Cache Size

The cache size of the proxy server is an important factor in determining the performance of the cache. Configuring too small a cache results in a low hit rate, which, in turn, translates to increased disk accesses and response times. Using a very large cache defeats the main purpose of reducing access time because the number of entries to check for a match increases. Therefore, an optimal cache size is essential.

The following table shows the hit rates of various cache sizes for 5, 50 and 100 simultaneous sessions:

Size/Sessions	**1M**	**10M**	**100M**	**1000M**
5	94.947%	95.045%	95.058%	95.065%
50	92.71 %	94.09%	95.03%	95.029%
100	90.90%	93.10%	94.93%	95.025%

Table 4.4. Cache Hit Rates

The table indicates that the hit rates for all cache sizes is above 90%. The high hit rate is because of the nature of SPECweb2005 webpages. Each webpage contains images and a large number of small static files. The hit rates differ by about 5% for the largest and small cache sizes. Since there is a constraint on the usable memory on the network processor, I have chosen 10M as the cache size for Squid in all the measurements.

4.2.2.2 Replacement Policy

The LRU (Least Recently Used) replacement policy is used in the Squid proxy server, since it is simple and robust to implement and uses the least processing resources.

4.2.3 System Metrics

The following metrics are considered in the benchmarks, to determine the effect of using the NP for task offloading:

1. Average CPU Utilization - Measures the average CPU utilization during the course of a run. When a system CPU is occupied by a process, it is unavailable for processing other requests. This becomes a bottleneck in the system. Therefore, a lower CPU utilization translates to high system availability. In this scenario, if the load on the Web server is less, it is available for servicing more requests.

2. Average Response Time (in seconds) - Measures the average latency encountered by the client in receiving a response. A shorter response time translates into a quicker response from the server, and fewer timeouts for the client.

3. Average Byte Rate - Measures the average bandwidth utilization of the link. A high ratio of ABR to available bandwidth indicates efficient utilization of the link.

4.3 Measurement Results

4.3.1 CPU Utilization

The graphs below indicate the average CPU utilization over the course of a run, for all three system setups. The values have been recorded for 100 simultaneous user sessions.

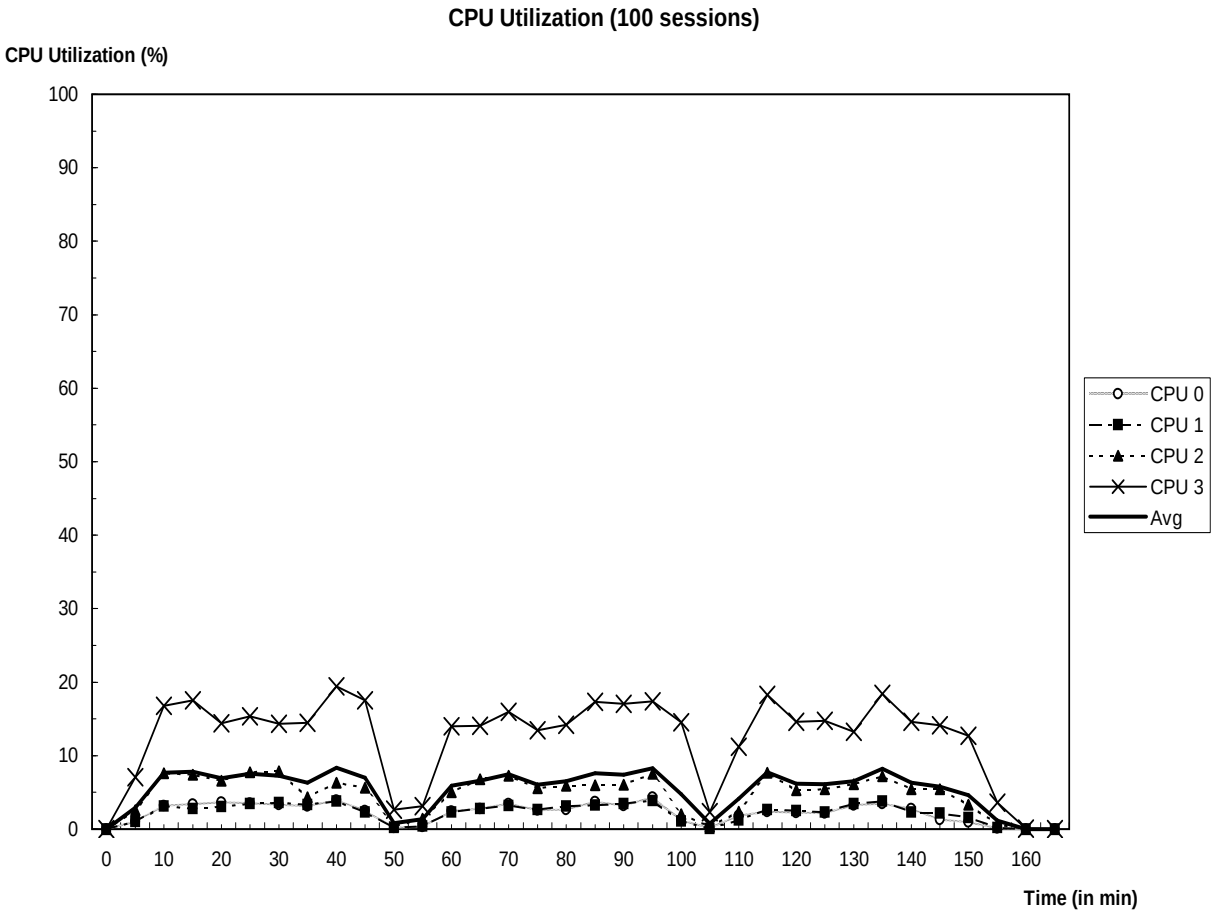

Figure 4.4. CPU Utilization - Web server only

CPU utilization measures the load on the system during the course of an entire run. The higher the load on the system, the higher the CPU utilization. The graphs indicate a pattern similar to the various phases in the benchmark run (fig. 4.2). The rampup, run and rampdown periods are all clearly visible from the graphs.

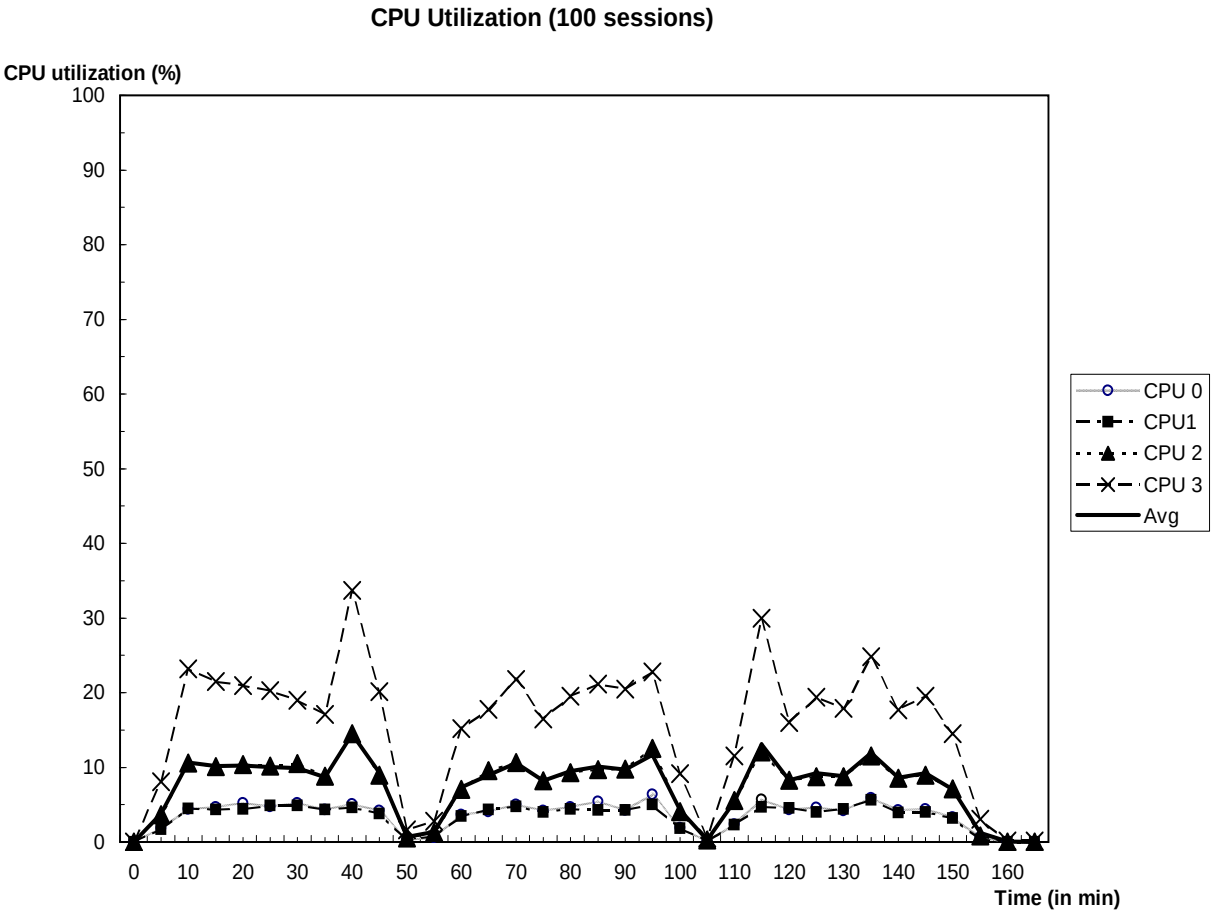

Figure 4.5. CPU Utilization - Web server and proxy

From the three graphs, a quantitative performance measurement can be made. The CPU utilization in 4.5 is higher than in 4.4. This is because the proxy server in 4.5 is installed on the same system as that containing the Web server. Therefore, this adds to the load on the system. The CPU utilization is lowest in 4.6, where the proxy server resides on the network processor. In this case, the proxy satisfies the requests and only directs the requests that were missing in the cache to the main Web server. Therefore, this reduces the load on the main server, since the proxy is implemented by a different processor (IXP2855).

The graphs also show that the CPU utilization is well below 100%, which indicates that the Web server can support more sessions. However, this system uses a

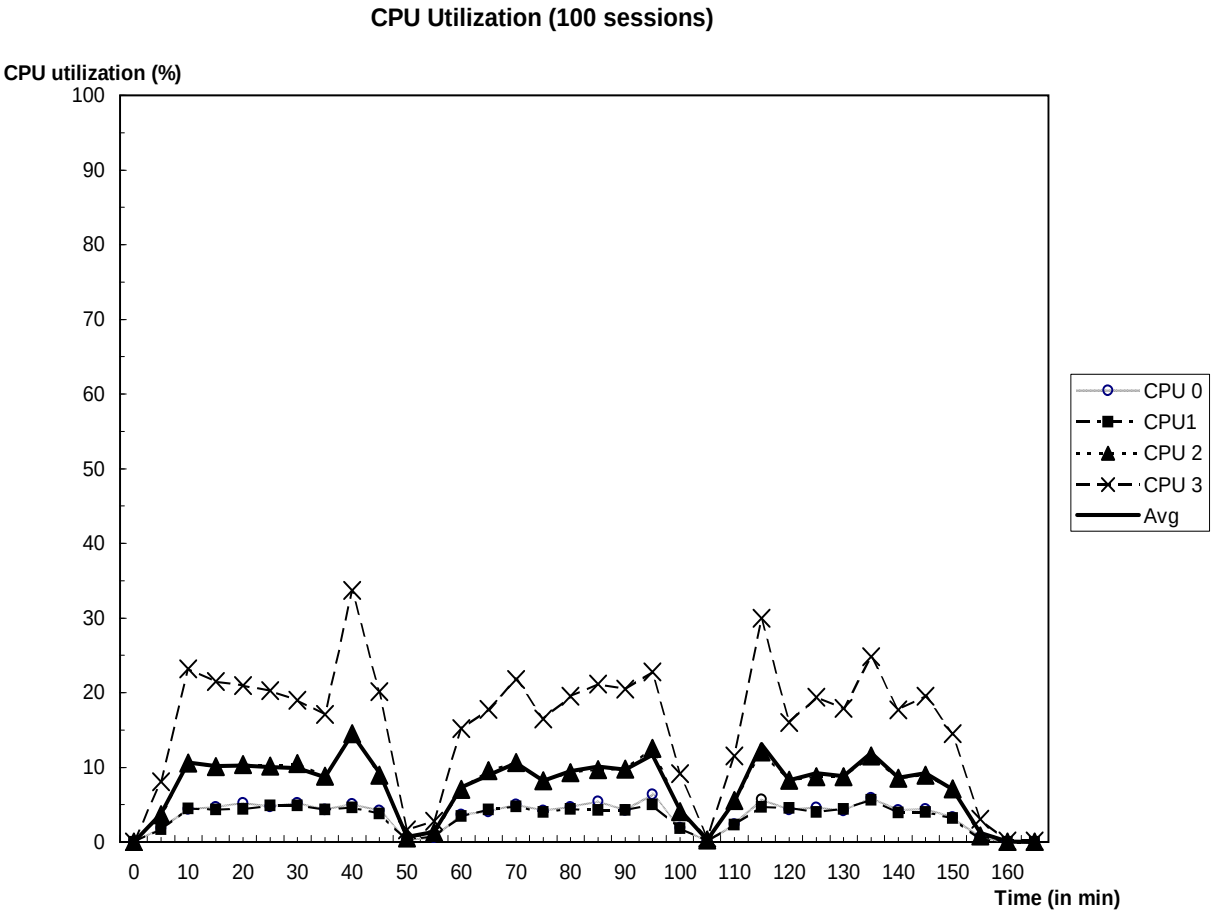

Figure 4.6. CPU Utilization - Web server and proxy on NP

single client, and therefore cannot handle over 100 sessions. An increased number of simultaneous sessions can be supported by increasing the number of clients.

4.3.2 Response Time

The graph below indicates the average response time for three different values of sessions - 5, 50 and 100, for the different system configurations.

Access times define how quickly (or slowly) a client receives a response once a request is made. Access times comprise of the following:

- Processing delay - Time taken by the Web server to process the request.
- Propagation delay - Time taken to travel the distance between the two systems.

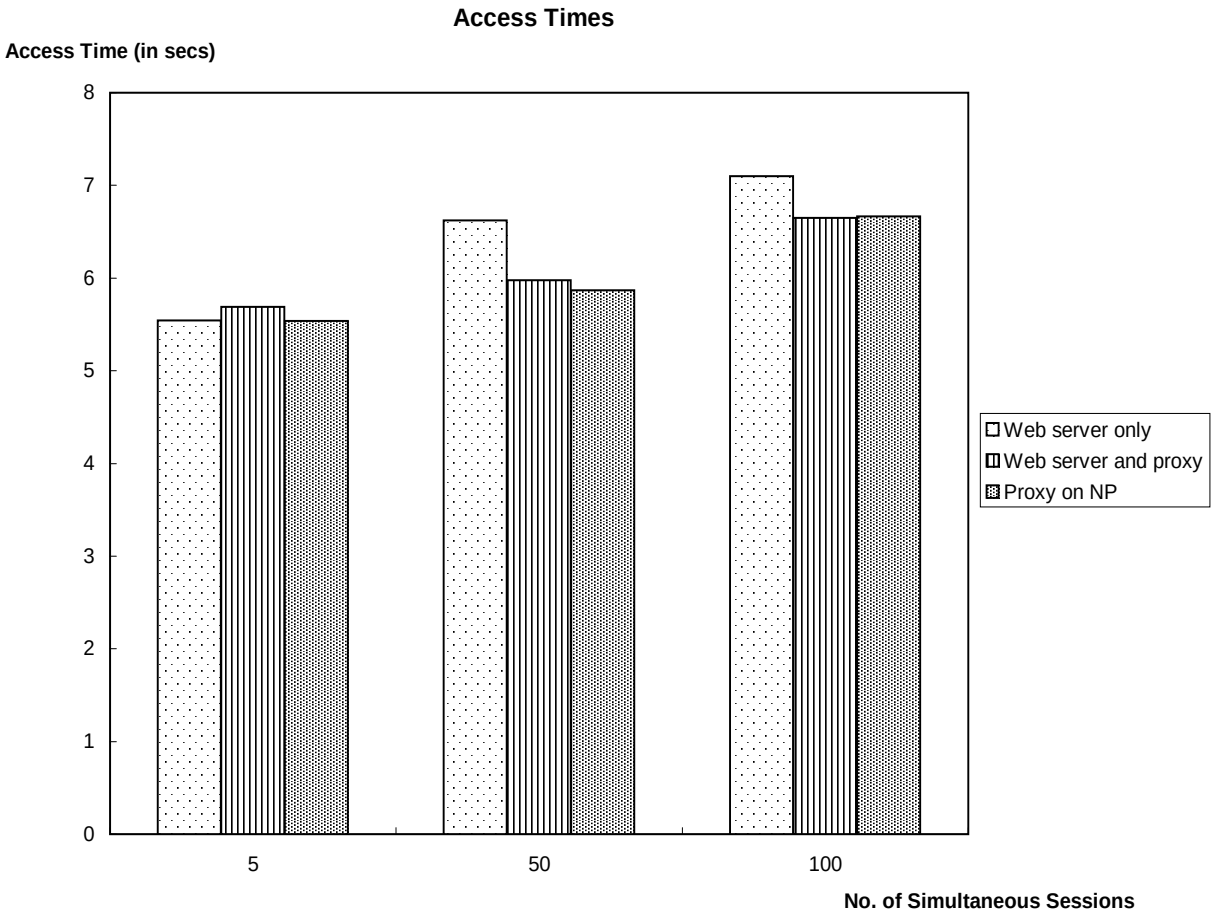

Figure 4.7. Access times for different configurations

- Queuing delay - Time taken to pass the queue, before the packet can be transmitted.
- Transmission delay - Time taken to transmit all of the packet's bits on the link.

Transmission delay is usually in the order of microseconds or less in practice. Propagation delay is not relevant in this system setup, since the two systems are in the same subnet. Therefore, these delays can be neglected. Therefore, processing and queuing delays are the main components of access time. Of these, queuing delays can be improved by implementing efficient queuing techniques. Processing delays are reduced by using the network processor to offload requests. Every request that requires static

objects, such as images, files etc. requires disk access, which is expensive. Caching frequently requested objects greatly reduces lookup times.

From the graph, it can be observed that the response times are relatively similar for all three configurations for low loads. The advantage of using proxy servers comes into play only at higher loads. For high loads, the response time for an IA-only system without a proxy is the largest. This is because the Web server handles all the requests itself, and takes longer to respond as more and more requests arrive, because of increased disk accesses. The response time for an IA system with a proxy server is lower than the previous case, because of the presence of the proxy server. The proxy caches frequently requested objects, thus avoiding the disk access time for obtaining the object. The response time for the IA/IXA system is the lowest, and therefore, implementing a proxy in the network processor improves the overall access time.

4.3.3 Average Byte Rate

The graph below indicates the average bytes/sec sent over the link for different session values and system configurations.

One of the important factors to be considered during performance measurement of a system is the efficiency of bandwidth utilization. There are two points to be noted:

- If the ratio of link utilization to available link bandwidth is high, then it indicates that the system is making efficient use of the bandwidth.
- The speed of data transfer is limited by the link speed. Even if the system is capable of supporting higher rates, the speed is still limited by the link bandwidth.

The graph indicates that all three system configurations make efficient use of the 100Mbps link. Between the three, the byte rate is lowest for the IA-only configuration without the proxy server. This is because only the Web server handles the requests

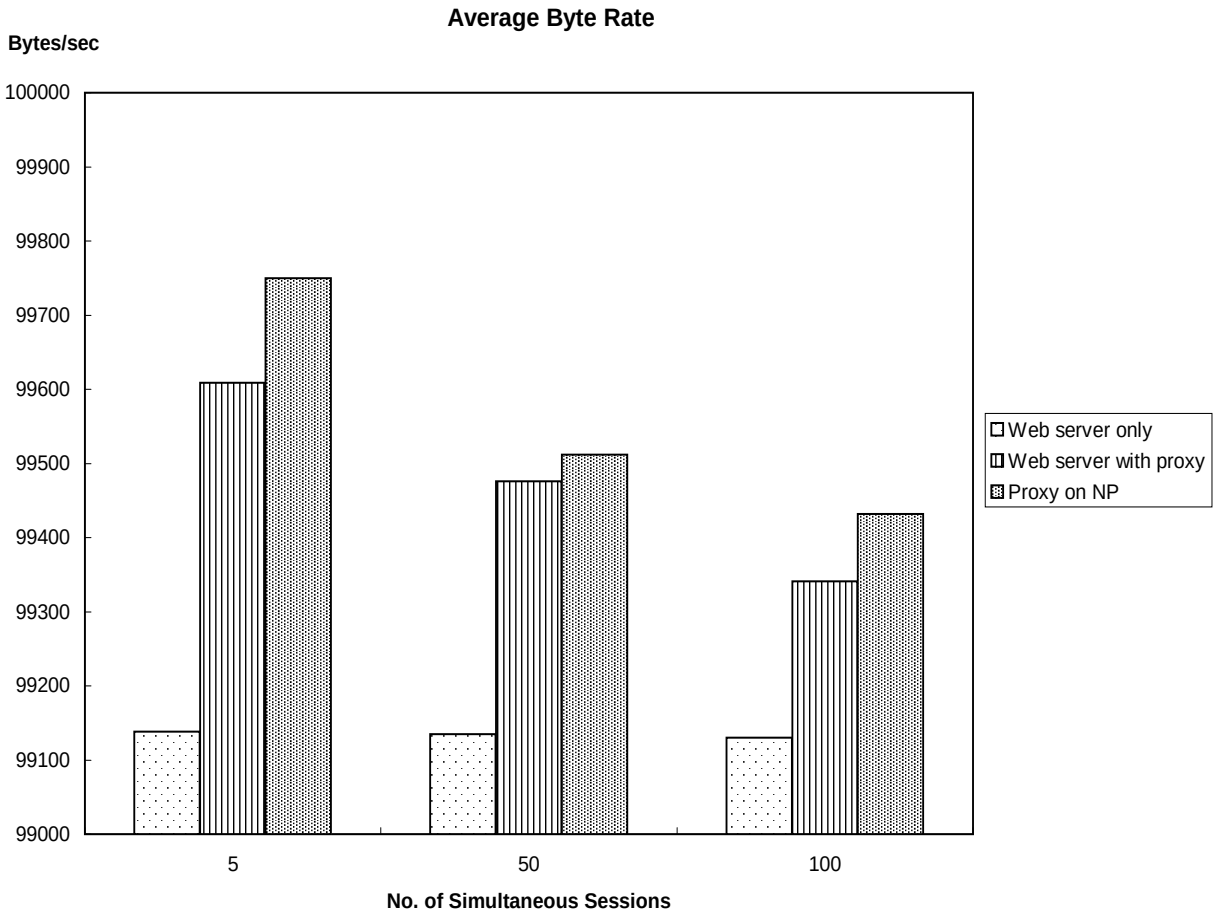

Figure 4.8. Average Byte Rates for different configurations

from the clients. Therefore, the number of requests it can serve per unit time frame is less, translating to a lower byte rate. The byte rate is higher in the IA configuration with the proxy server. Since the proxy server caches frequently requested objects, it can cater to more requests per time period, which in turn leads to a higher byte rate. The byte rate is the highest for the IA/IXA configuration. The proxy on the NP directs only the requests that it cannot serve to the main server, and serves the remaining requests itself.

4.4 Summary of Results

The following key deductions can be made from the results:

- The CPU utilizations indicated in 4.4, 4.5 and 4.6 indicate that using a proxy server reduces the CPU load to a large extent. Further, implementing the proxy server on the NP cuts the CPU utilization by over a half.

- The figure depicting access times in 4.7 also shows a reduction in latency when using a proxy in the network processor. The effect is more pronounced for higher loads.

- 4.8 shows maximum utilization of the link bandwidth when using a proxy server on the network processor.

4.5 Practical Implementation

There are two points to be noted when using such a system in a real-world scenario:

1. The delay between the NP and server is considered to be negligible (since they are located on the same system), and the delay between the server and client is also negligible (because they are located on the same subnet). Therefore, in a situation for which the above conditions are not true, the results may vary.

2. This model does not consider congestion caused due to regular Internet traffic. In a situation where the network is congested, response times could be much longer than those indicated in the graphs, and more erratic.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

I have introduced the idea of using a network processor for offloading applications in order to improve Web server performance. I have described the working of Web servers and proxy servers, and their performance parameters. I have described the reasoning behind application-level offloading, and the motivation behind choosing a network processor for this application. I have provided a detailed description of Web server performance improvement techniques that have been employed so far, and how this technique differs from them. A big advantage of this technique is that it is relatively easy for this architecture to be sold as an end-system. So the potential impact of this work is significant.

The results using a network processor show an improvement over the IA-only configurations. CPU utilization is reduced by over a half, latency times are reduced and link utilization is also more efficient. Some problems that may be encountered in real-world scenarios are also discussed.

5.2 Future Work

My thesis presents the idea of application-level offloading onto network processors, and the positive effect that it has on Web server performance. The following improvements, if made to the system, may considerably increase performance:

- Implementation in microengines - The implementation in this thesis makes use of the XScale processor in the IXP2855. This may be redesigned in a way to

make use of the microengines for the proxy implementation. Microengines are designed specifically for network-intensive applications, and can process packets at line speed. The IXP2855 has 16 mircoengines, which can considerably speed up data transfer.

- Extending Application Offloading - The concept of application offloading can be applied to other workloads. The IXP2855 has two cores for cryptographic applications, which implement DES and AES, and support a variety of key lengths. Each core operates independently, which allows simultaneous processing of multiple secure packets within each block. In addition, the ability to load cryptography keys while a block is simultaneously processing packets enables the network processor to support a large numbers of sessions. By processing data blocks as they arrive, the cryptography elements enable secure processing on the fly. This makes them efficient for security related workload processing, such as SSL/TLS or IPSec environments.

- Increased Number of Clients - The number of simultaneous sessions in this case is limited by the number of clients. A single client can get overloaded quickly. To determine the maximum number of sessions the different configurations can support, more clients can be added to the setup.

BIBLIOGRAPHY

[1] Andrew, Banks, Jim, Challenger, Paul, Clarke, Doug, Davis, Richard, P. King, Karen, Witting, Andrew, Donoho, John, Holloway Time Ibbotson, and Stephen, Todd. Httpr specification. Tech. rep., International Business Machines Corporation, 2002.

[2] Binkert, Nathan L., Saidi, Ali G., and Reinhardt, Steven K. Integrated network interfaces for high-bandwidth tcp/ip. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM Press, pp. 315–324.

[3] Cardellini, V., Colajanni, M., and Yu, P. S. Dynamic load balancing on web-server systems. *IEEE Internet Computing 3*, 3 (1999), 28–39.

[4] Cardellini, Valeria, Casalicchio, Emiliano, Colajanni, Michele, and Yu, Philip S. The state of the art in locally distributed web-server systems. *ACM Comput. Surv. 34*, 2 (2002), 263–311.

[5] Comerford, R. No longer in denial. *Spectrum IEEE 38*, 1 (2001), 59–61.

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

[7] Dierks, T., and Allen, C. Rfc 2246 the tls protocol version 1.0. Tech. rep., Network Working Group, 1999.

[8] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. "rfc 2616: Hypertext transfer protocol – http/1.1". *Internet Engineering Task Force* (June 1999).

[9] Gilmore, C., Kormann, D., and Rubin, A. D. Secure remote access to an internet web server. *Network. IEEE 13*, 6 (1999), 31–37.

[10] Institute, Informartion Sciences, and of Southern California, University. "rfc 793: Transmission control protocol". *Internet Engineering Task Force* (September 1981).

[11] Kant, K. Tcp offload performance for front-end servers. In *Global Telecommunications Conference 2003. GLOBECOM '03. IEEE* (2003), vol. 6, pp. 3242–3247 vol.6.

[12] Kargl, Frank, Maier, Joern, and Weber, Michael. Protecting web servers from distributed denial of service attacks. In *WWW '01: Proceedings of the 10th international conference on World Wide Web* (New York, NY, USA, 2001), ACM Press, pp. 514–524.

[13] Kreger, Heather. Fulfilling the web services promise. *Commun. ACM 46*, 6 (2003), 29–ff.

[14] Narasimha. Effectiveness of caching policies for a web server. In *High Performance Computing 1997. Proceedings. Fourth International Conference on* (1997), pp. 94–99.

[15] Netcraft. The netcraft web server survey. Tech. rep., Netcraft Web Site, 2007.

[16] Pandey, Raju, Barnes, J. Fritz, and Olsson, Ronald. Supporting quality of service in http servers. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1998), ACM Press, pp. 247–256.

[17] Squid. Squid caching proxy server. Tech. rep., Squid Web Site, 1996.

[18] Tatarinov, I., Rousskov, A., and Soloviev, V. Static caching in web servers. In *Proc. Sixth IEEE Intl. Conf. on Computer Communications and Networks* (1997), pp. 410–417.

[19] Wang, Jia. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev. 29*, 5 (1999), 36–46.

[20] Wessels, D., and Claffy, K. "rfc 2186: Internet cache protocol (icp) version 2". *Internet Engineering Task Force* (September 1997).

[21] Westrelin, R., Fugier, N., Nordmark, E., Kunze, K., and Lemoine, E. Studying network protocol offload with emulation: approach and preliminary results. In *12th Annual IEEE Symposium on High Performance Interconnects 2004 Proceedings* (2004), pp. 84–90.

[22] youb Kim, Hyong, Pai, Vijay S., and Rixner, Scott. Increasing web server throughput with network interface data caching. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ACM Press, pp. 239–250.

[23] youb Kim, Hyong, and Rixner, Scott. Tcp offload through connection handoff. In *EuroSys '06: Proceedings of the 2006 EuroSys conference* (New York, NY, USA, 2006), ACM Press, pp. 279–290.

[24] Zhao, Li, Luo, Yan, Bhuyan, Laxmi N., and Iyer, Ravi. A network processor-based content-aware switch. *IEEE Micro 26*, 3 (2006), 72–84.