Improving Web Server Efficiency on Commodity Hardware

by

Vicenç Beltran Querol

Advisors:

Jordi Torres i Viñals

Eduard Ayguadé i Parra

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR PER LA UNIVERSITAT POLITÈCNICA DE CATALUNYA

COMPUTER ARCHITECTURE DEPARTMENT (DAC)

TECHNICAL UNIVERSITY OF CATALONIA (UPC)

Barcelona, Spain

September 2008

ACTA DE QUALIFICACIÓ DE LA TESI DOCTORAL

Reunit el tribunal integrat pels sota signants per jutjar la tesi doctoral:

Títol de la tesi: Improving Web Server Efficiency on Commodity Hardware .....
Autor de la tesi: Vicenç Beltran Querol .....

Acorda atorgar la qualificació de:

- ☐ No apte
- ☐ Aprovat
- ☐ Notable
- ☐ Excel·lent
- ☐ Excel·lent Cum Laude

Barcelona, …… de/d'…..…….. de .....….

El President

.....
(nom i cognoms)

El Secretari

.....
(nom i cognoms)

El vocal

.....
(nom i cognoms)

El vocal

.....
(nom i cognoms)

El vocal

.....
(nom i cognoms)

Abstract

The unstoppable growth of the World Wide Web requires a huge amount of computational resources that must be used efficiently. Nowadays, commodity hardware is the preferred platform to run web server systems because it is the most cost-effective solution. The work presented in this thesis aims to improve the efficiency of current web server systems, allowing the web servers to make the most of hardware resources. To this end, we first characterize current web server system and identify the problems that hinder web servers from providing an efficient utilization of resources. From the study of web servers in a wide range of situations and environments, we have identified two main issues that prevents web servers systems from efficiently using current hardware resources. The first is the extension of the HTTP protocol to include connection persistence and security, which dramatically impacts the performance and configuration complexity of traditional multi-threaded web servers. The second is the memory-bounded or disk-bounded nature of some web workloads that prevents the full utilization of the abundant CPU resources available on current commodity hardware. We propose two novel techniques to overcome the main problems with current web server systems. Firstly, we propose a Hybrid web server architecture which can be easily implemented in any multi-threaded web server to improve CPU utilization so as to provide better management of client connections. And secondly, we describe a main memory compression technique implemented in the Linux operating system that makes optimum use of current multiprocessor.s hardware, in order to improve the performance of memory bound web applications. The thesis is supported by an exhaustive experimental evaluation that proves the effectiveness and feasibility of our proposals for current systems. It is worth noting that the main concepts behind the Hybrid architecture have recently been implemented in popular web servers like Apache, Tomcat and Glassfish.

Agraïments

Aquesta tesi no hagués estat possible sense el suport de molta gent, especialment de la meva família i amics, que han estat sempre al meu costat. Vull agrair a Gemma el temps que hem compartit durant aquest anys i també, tot el que ens resta encara per viure.

També vull agrair a tot els meus company els bons moments que hem passat, que han fet que aquest darrers anys passessin tan ràpidament. No podria acabar el agraïments sense mencionar al David Carrera, amb el que he rigut molt i del que encara he après més.

Finalment, un agraïment especial als meus directors de tesi, Jordi Torres per transmetrem la seva passió per la recerca i els nous reptes i Eduard Ayguadé que sempre m'ha motivat per arribar més lluny.

Contents

List of Figures

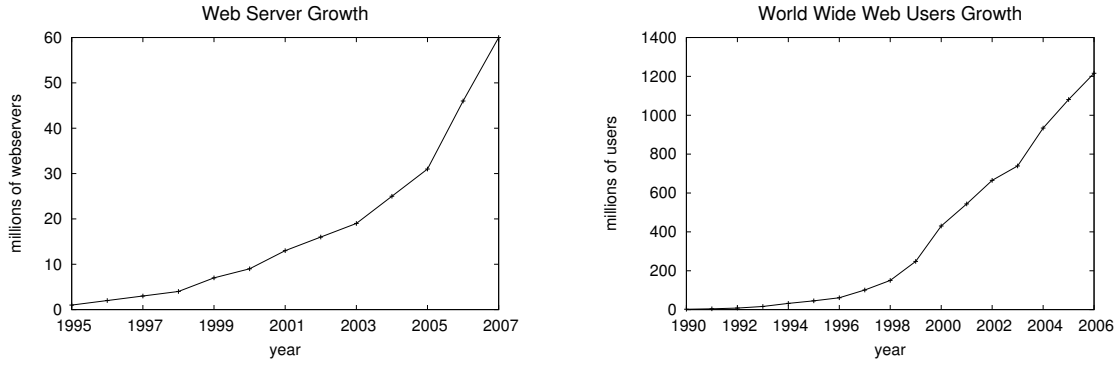
List of Tables

Chapter 1

Introduction

The World Wide Web (*WWW*) has experienced an astonishing evolution since its creation. In its early days, the *WWW* was composed of only a thousand simple web servers that provided static files to text-based browsers through the original stateless HTTP protocol. With the arrival of Mosaic [63], the first popular graphical browser, the size of the web started to grow rapidly, as we can see in figure 1.1, which shows the growth trend of both the number of web servers and users. With the popularization of the *WWW*, the complexity and requirements of web applications has also increased. To cope with these new requirements and complexities, the hardware and software behind web applications have also evolved as we can see in Figure 1.2. **This rapid growth in the size and complexity of the *WWW* has multiplied the computational resources needed to provide the ever increasing quality of service that web users expect. The overall industry has shifted from high performance specialized hardware to clusters of low cost commodity hardware, -as this is the only cost-effective solution to providing the huge computational power required by web servers.**

From the early days of the Internet, server side techniques used to develop web applications have been incrementally subsumed by more powerful technologies which simplify the development of complex web applications. Nowadays, the most representative technologies of this tendency are the Java and .Net platforms that are bundled with a lot of standard API's to perform tasks ranging from the simplest socket write to the most complex database query. **These technologies have changed the web servers from simple servers of static files to complex application servers that dynamically produces the required content from back-end systems.** The HTTP protocol has also been extended to accommodate new web application requirements such as stateful session



(a) Growth in the number of web server [33]

(b) Growth in the number of web users [64]

Figure 1.1 World Wide Web Growth

with connection persistence or security through encryption techniques.

In contrast to the continuous evolution of web technologies and communication protocols, the initial multi-threaded web server design has remained unchanged. The only alternative to the well known multi-threaded architecture is the event-driven architecture, and this has only been adopted for specific web server uses. The difference between the two main architectural options for a web server design lies in the concurrency programming model chosen for implementation. The two major alternatives are the original multi-programmed model and the more recent event-driven model[67] [85].

The multi-programmed model has different names depending on the execution abstraction used, -such as multiprocess or multi-threaded mode. In this text, we use the multi-threaded approach based on a pool of threads, which is widely used on current web servers. In both event-driven and multi-threaded models the jobs to be performed by the server are divided into work assignments that are each assumed by a worker thread. If a multi-threaded model is chosen, the unit of work that can be assigned to a worker thread is a client connection. This is achieved by creating a virtual association between the thread and the client connection and is not broken until the client closes the connection or a keep-alive timeout expires. Alternatively, in an event-driven model, the work assignment unit is a client request, so there is no real association between a server thread and a client. **The event-driven architecture is more efficient than the multi-threaded one because it can serve a large number of clients with a small number of threads** thanks to its ability to keep the clients' connection open without requiring one active thread for each connection. This operation model avoids worker threads from being kept blocked in socket read operations during client think times and eliminates the need to introduce connection inactivity timeouts and their associated drawbacks. Additionally, as the number of worker

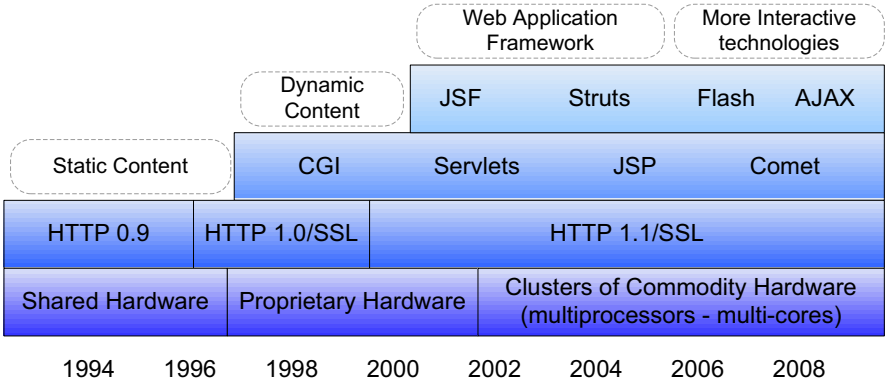


Figure 1.2 Evolution of Web Server Technologies

threads can be very low (one per CPU should be enough) contention inside the web container is reduced. Furthermore, **the complexity of the event-driven architecture is high and grows quickly when the web server needs to support dynamic technologies**, like JSP or PHP, or needs to be integrated with back-end servers, because the non-blocking nature of the worker threads do not fit well with these approaches. Therefore, **the multi-threaded architecture is the only one used in complex applications servers because it is easier to implement and integrate with legacy back-end servers which also follow a multi-threaded design. In contrast, event-driven servers are usually used only as high-performance web servers for cached static content.**

From the study of both architectures in a wide range of situations and environments, **we have identified two main issues that prevent web servers systems from efficiently utilizing current hardware resources. The first is the extension of the HTTP protocol to include connection persistence and security**, which dramatically impacts the performance and configuration complexity of traditional multi-threaded web servers. **The second is the memory-bounded or disk-bounded nature of some web workloads** which prevents the full utilization of the abundant CPU resources available on current commodity hardware. **To overcome these problems we propose two novel techniques described in the next sections. The first is a new Hybrid web server architecture aimed at solving the current limitations of multi-threaded web servers. The second is a main memory compression technique implemented in the operating system which improves the performance of memory-bounded web applications.**

Hybrid Architecture

The introduction of connection persistence in the HTTP/1.1 protocol had a a dramatic performance impact on existing multi-threaded web servers. Persistent connections (i.e.: connections that are kept alive by the client between two successive HTTP requests and can in turn be separated in time by several seconds of inactivity (think times)), can cause many server threads to be retained by clients even when no requests are being issued, so the thread is kept in an idle state. The use of blocking I/O operations on the sockets is the cause of this performance degradation scenario. The situation can be solved by increasing the number of threads available (which in turn increases the contention for shared resources in the server which require exclusive access) or introducing an inactivity timeout for established connections, which forces a client's connection to close when the keep-alive timeout expires, with the consequent loss of the benefits of persistent connections. With the introduction of secure connections with the HTTP/s protocol, this problem is exacerbated, because, in this case, the establishment of a new connection is

heavily penalized due to the computational cost of the cryptographic techniques involved in the handshake. Another problem introduced by the connection persistence is that, the multi-threaded web server has two key parameters that influence performance; the keep-alive timeout and the number of threads. The difficulty in understanding their impact on performance stems from the fact that their interaction also depends on the application's workload type. These considerations explain the poor web server performance which fails to utilize all the available CPU resources.

The Hybrid architectures (discussed in Chapter 4, below) can take advantage of the strong points of both multi-threaded and event driven architectures. In a Hybrid architecture, the operation model of the event-driven architecture is used to assign client requests to worker threads (instead of client connections), and the multi-threaded model is used to process client requests, where the worker threads perform blocking I/O operations when required, but only when new data is available. This architecture can be used to decouple the management of active connections from the request processing and servicing activity of the worker threads. In this way, the web container logic can be implemented following the natural multi-threaded programming model and the management of connections can be done with the highest performance possible, without blocking I/O operations and therefore achieving a maximum overlapping of the client think times with the processing of requests. In this architecture the role of the acceptor thread, as well as the request dispatcher role, is maintained from the pure event-driven model, and the worker thread pool (performing blocking I/O operations when necessary) is obtained from the pure multi-threaded design. This makes it possible for the Hybrid model to avoid closing connections to free worker threads (with the keep-alive timeout) without renouncing the multi-threaded paradigm. Consequently, the Hybrid architecture makes better use of the characteristics introduced by the HTTP/1.1 protocol, such as connection persistence, avoiding dispensable client re-connections with only a low number of threads. If the HTTP/s protocol is used, closing connections, especially when the server is overloaded, increase the number of cryptographic handshakes performed and drastically reduces the capacity of the server, which has an important impact on the maximum throughput that can be achieved by the web server. In addition, web server configuration complexity is greatly reduced because the Hybrid architecture only has one configuration parameter -the number of threads- which does not depend on the web workload type. Finally, this Hybrid architecture can be easily implemented in any existing multi-threaded web server, and it is thus a feasible way to improve current web server systems.

Main Memory Compression

Current web servers are highly multi-threaded applications whose scalability benefits from the current multi-core/multiprocessor trend. However, some workloads can not capitalize on this because their performance is limited by the available memory and/or disk bandwidth, which prevents the server from taking advantage of the computing resources provided by the system. To solve this situation we propose the use of main memory compression techniques which increment the available memory and mitigate the disk bandwidth problem, allowing the web server to improve its use of CPU system resources. The objective of main memory compression techniques is to reduce the in-memory data size to virtually enlarge the available memory on the system. The main benefit of this technique is the reduction of slow disk I/O operations, which improves data access latency and save disk I/O bandwidth. With our implementation of this technique in the Linux operating system, we have obtained promising results, such as a 30% web server throughput improvement, and a 70% disk access reduction on a multiprocessor system. On the other hand, its main drawback is the large amount of CPU power needed by the computationally expensive compression algorithms, and this makes it unsuitable for medium to large CPU intensive web applications. For this reason, in Chapter 5 below the use of these techniques on the Cell processor, a heterogeneous multi-core design where the computationally expensive compression and decompression tasks are accelerated in specialized coprocessors.

1.1 Thesis Contributions

The contributions of this thesis are discussed in Chapters 3, 4 and 5. Chapter 3 deals with a deep study of multi-threaded web server architecture, even-driven web server architecture, the scalability of the Java platform, and the impact of the HTTP and HTTP/S protocol on web server performance. It also introduces the configuration complexity problem of multi-threaded architecture. The main conclusions of this study are that the multi-threaded architecture is unable to make the most of current CPU resources, specially under secure workloads. There are three publications related to the study presented in this chapter:

[20] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. **Evaluating the scalability of java event-driven web servers**. *In ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing*, pages 134–142, Montreal, Quebec, Canada, 15-18 August 2004.

[22] V. Beltran, J. Guitart, D. Carrera, J. Torres, E. Ayguadé and J. Labarta.

Performance Impact of Using SSL on Dynamic Web Applcations. *In Jornada '04: Proceedings of the 15th Jornadas de Paralelismo*, pages 471-476, Almeria, Spain, 15-17 September 2004.

[41] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. **Characterizing secure dynamic web applications scalability**. *In IPDPS '05: Proceedings of the 19th International Parallel and Distributed Processing Symposium*, pages 108–116, Denver, CO, USA, 04-08 April 2005.

Chapter 4 presents a new Hybrid web server architecture which combines the advantages of multi-threaded architecture and the performance and scalability of event-driven architecture. This Hybrid architecture improves the throughput and makes the most of the available CPU resources, -while dramatically reducing the configuration complexity of multi-threaded web servers. The study also makes it clear that memory-bounded and disk-bounded web applications under-utilize the available CPU resources of current commodity hardware. Studies of the Hybrid architecture were presented in the following publications:

[27] D. Carrera, V. Beltran, J. Torres, and E. Ayguadé. **A hybrid web server architecture for e-commerce applications**. *In ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems*, pages 182–188, Fuduoka, Japan, 20-22 July 2005.

[19] V. Beltran, D. Carrera, J. Guitart, J. Torres, and E. Ayguadé. **A hybrid web server architecture for secure e-business web applications**. *In HPCC '05: Proceedings of the 1st International Conference on High Performance Computing and Communications*, pages 366–377, Sorrento, Italy, 21-22 September 2005.

[23] V. Beltran, J. Torres, and E. Ayguadé. **Understanding tuning complexity in multithreaded and hybrid web servers**. *In IPDPS '08: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, pages 56–64, Miami, FL, USA, 14-18 April 2008.

Finally, Chapter 5 proposes a main memory compression technique, implemented in the Linux operating system, that make the most of current multiprocessor hardware, thereby improving the performance of memory-bounded or disk-bounded web applications. Chapter 5 also discusses an asynchronous implementation of the memory compression used in the Cell processor, a heterogeneous multi-core design, where the compression and decompression are accelerated in the coprocessor units. Our studies on main memory compression techniques are outlined in these publications:

[25] V. Beltran, J. Torres, and E. Ayguadé. **Improving disk bandwidth-bound applications through main memory compression**. *In MEDEA '07: Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications, systems and architecture*, pages 57–63, Brasov, Romania, September 2007.

[24] V. Beltran, J. Torres, and E. Ayguadé. **Improving web server performance through main memory compression**. *In ICPADS '08: Submitted to the 14th International Conference on Parallel and Distributed Systems*, Melbourne, Victoria, Australia, December 2008.

To summarize, this thesis provides a deep analysis of current web server systems and proposes two novel techniques aimed at improving the efficiency of web servers, which are implemented in current systems and validated by strong experimental results. The experimental environment that has was used in this thesis is summarized in Figure 1.3. The environment is divided into three main hardware components: the web server system, the interconnection network and the web clients. Our study is focused on the web server system, which is divided into several layers: the web application, the web server and platform, the operating system and the hardware layer. The following publications are also related to the work presented in this thesis:

[45] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguadé. **Dynamic CPU Provisioning for Self-Managed Secure Web Applicatios in SMP Hosting Platforms**. *Computer Networks, Vol. 52 (7), pp. 1390-1409, ISSN: 1389-1286*, May 2008

[44] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguadé. **Designing an Overload Control Strategy for Secure e-Commerce Aplications**. *Computer Networks, Vol. 51 (15), pp. 4492-4510, ISSN: 1389-1286*, October 2007

[43] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. **Session-Based Adaptative Overload Control for Secure Dynamic Web Aplication**. *In ICPP '05: The 2005 International Conference on Parallel Processing*, Oslo, Norway. ISBN 0-7695-2380. ISSN 0190-3918., June 2005

1.2 Thesis Organization

The thesis is structured as follows: Chapter 2 introduces the hardware environment, web benchmarks and the methodology used throughout this thesis to evaluate the experimental results. Chapter 3 characterizes web servers behavior, scalability and performance on commodity hardware and identifies the problems and bottlenecks associated with

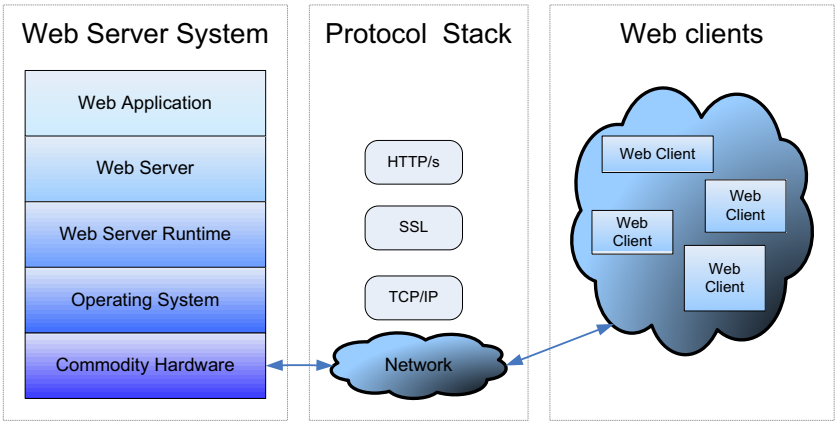


Figure 1.3 Generic Web Server Environment

different workloads and communication protocols. In Chapter 4 the Hybrid architecture is described, its performance and scalability are evaluated in different scenarios and web workloads. The configuration complexity of both multi-threaded and Hybrid web servers are compared. Chapter 5 describes the main memory compression technique for symmetric multiprocessor and heterogeneous processor systems, whose function is to improve the performance of memory-bounded and disk-bounded web workloads. Finally, Chapter 6 contains conclusions and proposals of future work.

Chapter 2

Methodology

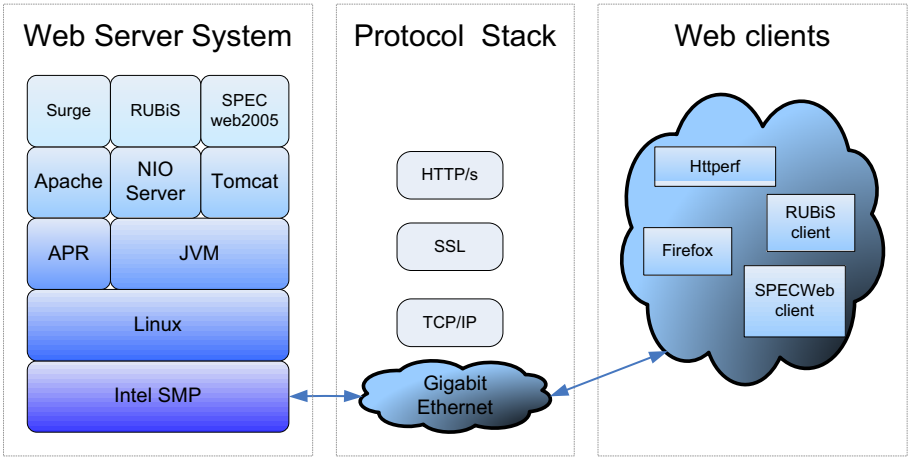


Figure 2.1 Web Server Environment

This thesis is based on experimental results obtained from the study and evaluation of realistic web server environments. All the measures and experiments are carried out with common hardware and well known software. Moreover, the two novel techniques proposed in Chapters 4 and 5 aimed at improving web server systems efficiency are also implemented and evaluated on currently-used systems. In the rest of this chapter the software and hardware environments used in this thesis are described.

The basic environment used to study web server systems are depicted in Figure 2.1. This diagram shows the main components needed to study and evaluate a web server system. First we have the "Web Server System" , which is composed of a web application (Surge, RUBiS or SPECweb2005), the web server (Apache Httpd, NIO prototype or Tomcat) and the operating system that runs on top of the hardware. Another important component is the "Web client" that is composed of one or more client machines that run a web client emulator software package(Httperf, RUBiS client emulator, etc) to mimic a group of web users. Finally, a network is needed to inter-connect the web server system and the web clients. The most basic method used to evaluate a web server system is to setup a web application benchmark and then measure its performance form the web client emulator.

Two hardware configurations for the web server system were used in this study; one based on a multiprocessor architecture from Intel and the other based on a multi-core/multiprocessor architecture from IBM. The web clients emulators run on a set of machines usually connected to the web server through a gigabit Ethernet switch. Three main web benchmarks were used (Surge, RUBiS and SPECweb20059 in several different environments and configurations. In the next sections, hardware, operating system, web server and web benchmarks components are described in detail.

2.1 Hardware

There are two main hardware platforms used to run the web server software. The first platform is based on a 4-way 1.4Ghz Xeon board from Intel with 2GB of RAM and a gigabit Ethernet card. The other platform is a 2-way 2.3Ghz PPC970MP (dual core) with 8GB of RAM and two gigabit Ethernet cards. Both SMP architectures are representative of current commodity hardware. The Httperf web client emulator usually run on a 2-way 2.4Ghz Xeon with 2GB of RAM, while the SPECweb client emulator run on a set of three multiprocessor OpenPower machines from IBM. The server and the web client emulators are always interconnected with a gigabit switch, unless otherwise specified. RUBIS and SPECweb2005 benchmarks need an additional machine to run a database (or an emulator of a database), which is also connected to the main gigabit switch. The experiments of section 5.6 were performed in a QS21 blade with two cell processors and 1GB of RAM.

2.2 Operating System and Web Server Platforms

The operating system used throughout this thesis is a system based on a Linux kernel. We have used several Linux distributions, including Red-Hat, Debian or SUSE. We have used two versions of the Java platform: the 1.4.2 and the 1.5 from SUN and IBM respectively.

2.2.1 Java Virtual Machine

The Java Virtual Machine is the environment used to execute the bytecode of Java programs. This environment provides all the features needed to build a web server from multi-threaded capabilities to network support. The performance of this platform in I/O intensive applications has been greatly improved with the NIO (New I/O) API, which provides non-blocking operations and other sets of advanced features currently available on most operating systems.

2.2.2 Apache Portable Runtime

The Apache Portable Runtime is aimed to isolate the Httpd web server from the underlying operating system. This Runtime provides basic and advanced I/O capabilities of the Operating System in a uniform way. This runtime provides a set of features equivalent to the NIO package of the Java platform.

2.2.3 Linux

Linux is a popular UNIX-like operating system. We have performed all the experiments evaluation on the 2.6 branch of this operating systems. The open source nature of Linux has allowed us to modify it so as to implement our main memory compression system described in Chapter 5. A common issue with this operating system is the rapid evolution of the 2.6 branch, which makes it difficult to compare performance results between different releases.

2.3 Web Servers

In this thesis three web server were used: the popular Apache Httpd 2.0 native web server, as a representative multi-threaded web server of static content, a NIO web server prototype to evaluate the performance of a event-driven web server of static content, and finally, the Tomcat web server, as a representative example of a powerful platform to build dynamic content and complex web application servers.

2.3.1 Apache Httpd 2.0

Apache Httpd is a popular web server that supports several programming models including the multi-process and multi-threaded architectures. In this thesis only the multi-threaded flavor was evaluated, as it is the most commonly used. The main characteristics of this type of web server are flexibility, security, reliability and performance.

2.3.2 NIO prototype

The 1.4 release of the Java 2 Standard Edition includes a new set of I/O capabilities created to improve the performance and scalability of intense I/O applications. The classic Java I/O mechanisms do not offer the level of efficiency necessary to run applications with high I/O requirements. Some widely existing I/O capabilities implemented on most of operating systems (such as mapping files to memory, readiness selection of channels and non-blocking I/O operations), were not supported by previous versions of the J2SE and have been included in the 1.4 release. The NIO package includes a readiness selection operation on channels (SocketChannels and FileChannels). This selection operation is used in the implementation of the main loop to detect which web clients have sent requests to the server or which sockets have data available. The non-blocking features of the NIO API have been used to create our web server core, based on the event-driven strategy,

which is used to evaluate the performance of a server based on this technology. Our web server prototype will be referred to as the NIO web server from now on.

2.3.3 Tomcat

Tomcat is a popular web server that can run as a stand-alone or embedded inside a J2EE server such as JBoos or Geronimo. Like the Apache web server, Tomcat is a modular web server which supports the multi-threaded and the Hybrid threading model, but does not support the pure event-driven one because this threading model does not fit well with the blocking semantics of the Servlet and JSP technologies. The standard Tomcat web server is composed of three major components: the Coyote connector, the Catalina container and the Jasper JSP compiler. The Coyote connector is in charge of accepting and managing clients' connections with a specific threading model. The Catalina container and Jasper compiler share a multi-threaded architecture and provide the Servlet and JSP functionalities of the web server. A remarkable fact is that the same threads that manage the connections and parse the HTTP requests on Coyote also execute the code of the Catalina container and the JSP compiler. In the next subsections, the three versions of the Coyote connector currently available are described: the original Coyote which implements a multi-threaded architecture and the Hybrid and APR versions which implement the Hybrid architecture. Each one of the connectors can be configured to manage plain connections as well as secure connections.

Multi-threaded Connector

The original Tomcat implementation of the Coyote connector follows a pure multi-threaded approach to manage the client connections. For each incoming connection, one thread is assigned to serve the requests until the client closes it or a keep-alive timeout occurs. The two key configuration parameters of this connector are the number of threads and the keep-alive timeout value because both parameters determine the performance of the web server [78][35]. Coyote implements a best effort heuristic to avoid low performance due to mis-configured keep-alive timeouts and number of threads parameters. This heuristic dynamically reduces the keep-alive timeout when the worker threads get busy.

Hybrid Connector

Our Hybrid connector is a modification of the original Coyote connector; although both connectors share most of the code (such as the HTTP request parsing code or the pool

of threads). The Hybrid connector follows the architecture design explained in chapter 4. The implementation of the Hybrid connector is feasible thanks to the Non Blocking I/O (NIO) API [50] that appeared in Java version 1.4, and provides the select() call that allows the thread-per-connection paradigm of the multi-threaded architecture to be broken. The Hybrid connector has only one configuration parameter, namely the number of worker threads, because the keep-alive timeout parameter is always set to infinite. For a more detailed description of the Hybrid connector implementation see Chapter 4.

APR Connector

The APR and Hybrid connectors share the same Hybrid architecture concept. The major difference between the two connectors is that the Hybrid connector is based on the Java NIO API, whereas the APR connector is based on the native Apache Portable Runtime [2] and uses it to perform native select system calls, which are available in most operating systems. In this thesis, we do not evaluate the performance of the APR connector as it is still under development. However, the preliminary tests of the APR connector show performance similar to the Hybrid connector and do not show any significant performance improvements on our systems.

2.4 Web Benchmarks

2.4.1 Httperf

The program used to generate the web workload for severali of the experiments is Httperf[62]. This workload generator and web performance measurement tool allows the creation of a continuous flow of HTTP requests issued from one or more client machines and processed by one Web server machine. The configuration parameters of the benchmarking tool used for the experiments presented in this thesis were set to create a realistic workload, with non-uniform reply sizes, and to sustain a continuous load on the web server. One of the parameters of the tool is the number of emulated clients on the client machine. Each emulated client issues a number of requests, some of them pipelined, over a persistent HTTP connection. The workload generated by Httperf can be specified in a trace file with a sequence of request and think-times. Httperf is usually utilized as an efficient replacement of Surge and RUBiS web client emulators.

2.4.2 Surge

The web access distributions produced by Surge are based on the observation of some real web server logs, from which was extracted a data distribution model of the observed workload. This method guaranties than the used workload for the experiments follows a realistic load distribution. A static content application is characterized by the short length of user sessions as well as by the low computational cost of each request to be serviced. This is the scenario reproduced with the Surge[17] web workload.

2.4.3 RUBiS

RUBiS [15] is a dynamic web application, that is characterized by the long length of user sessions (an average of 300 requests per session, in contrast to the 6 requests per session for the static Surge workload) as well as by the high computational cost of each request to be serviced (including embedded requests to external servers, such as databases). RUBiS implements the core functionality of an auction site: selling, browsing and bidding. The RUBiS client emulator uses a Markov model to determine the access pattern to the web application. RUBiS defines 27 interactions. Among the most important ones are browsing items by category or region, bidding, buying or selling items and leaving comments on other users. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. RUBiS supplies implementations using some mechanisms for generating dynamic web content like PHP, Servlets and several kinds of EJB. In this thesis we focused on the Servlet implementation of this benchmark. The configuration parameters of the benchmarking tool used for the experiments presented in this thesis were set to create a realistic workload, with non-uniform reply sizes, and to sustain a continuous load on the server. One of the parameters of the tool represents the number of concurrent clients interacting with the server. Each emulated client opens a session with the server. The session remains alive for a period of time, called session time, at the end of which the connection is closed. Each session is a persistent HTTP or HTTP/S connection with the server. Using this connection, the client repeatedly makes a request (the client can also pipeline some requests), parses the server response to the request, and follows a link embedded in the response. The workload distribution generated by Httperf was extracted from the RUBiS client emulator, which uses a Markov model to determine which subsequent link from the response to follow. Each emulated client waits for an amount of time, called the think time, before initiating the next interaction. This emulates the thinking period of a real client who takes a period of time before clicking on the next

request. The think time is generated from a negative exponential distribution with a mean of 7 seconds. Httperf also allows the user to configure a client timeout. If this timeout elapses and no reply has been received from the server, the current persistent connection with the server is discarded, and a new emulated client is initiated.

2.4.4 SPECWeb2005

The substantial differences in the security requirements and dynamic content of various web server workload types make it impossible to synthesize them into a single representative workload. For this reason SPECweb2005 [9] contains three workload applications that attempt to represent three different types of real world web applications.

The first application is the Support workload, based on a web application for downloading patches and tests the ability to download large files over plain connections. The two principal characteristics of the Support application are the use of only plain connections and a large working set per client, so the benchmark tends to be I/O disk intensive. The second is the Banking workload that imitates an online banking web application. The principal characteristic of the Banking workload is the use of secure connections only, where all the transmitted data is encrypted / decrypted, so the benchmark becomes CPU intensive. The last is the E-commerce workload, based on an e-commerce web application. The principal characteristic of the E-commerce application is the mix of secure and plain connections to mimic the behavior of real e-commerce sites. Usually a client navigates through the application looking for a product over a plain connection and when a product is selected, the client changes to a secure connection in order to buy the product. The number of plain connections is usually greater than the secure ones and the working set per client is quite large, so the benchmark balances the CPU and the I/O disk usage, but tends to be more CPU intensive. The three workloads are based on requests for web pages and involve running a dynamic script on the server end and returning a dynamically created file, followed by requests for embedded static files.

The benchmark client emulator is based on a the activity of number of customers who are concurrently navigating the application, switching from active to passive states and vice versa. A client in the active state performs one or more requests to the web server to simulate user activity, whereas a client in the passive state simulates a user think-time between active states. The time spent in passive states (or think-times) are distributed following a geometric distribution. For each workload type a Markov Chain is defined to model the client access pattern to the web application and a Zipf distribution is used to access each web application's working set.

SpecWeb2005 also has three other important features which help to mimic a real

environment. Firstly, it includes the use of two connections for each client to perform parallel requests. Secondly, it simulates browser caching effects by using If-Modified-Since requests. Finally, the web server application working set size is proportional to the number of clients simulated.

2.5 Tools

In order to evaluate our web server systems we use the metrics obtained by the web client emulator as well as system tools that provides information about the system state such as top or vmstat. Moreover, to study in great detail some web server environments we use a performance analysis framework to analyze the web server behavior. This framework, which consists of an instrumentation tool called Java Instrumentation Suite (JIS [28]) and a visualization and analysis tool called Paraver [68], considers all levels involved in the application server execution (operating system, JVM, application server and application), allowing a fine-grain analysis of dynamic web applications. For example, the framework can provide detailed information about thread status, system calls (I/O, sockets, memory and thread management, etc.), monitors, services, connections, etc.

Chapter 3

Web Servers Systems Characterization

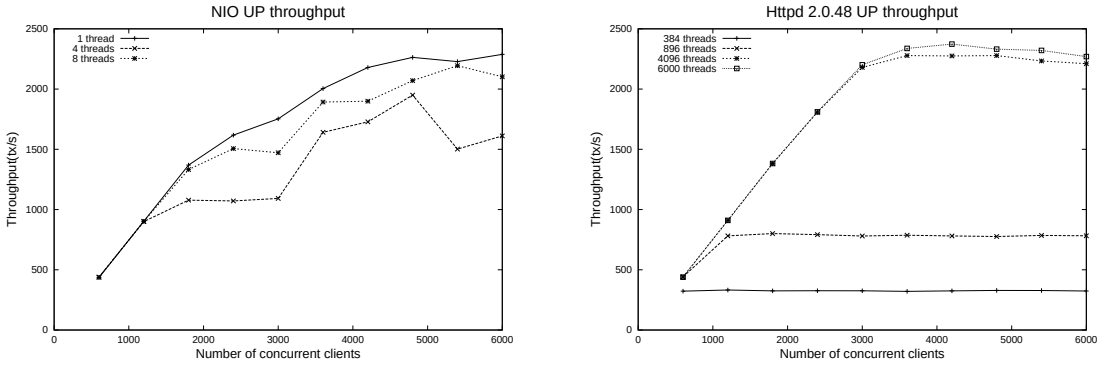
This chapter begins with a description of the main characteristics of both the multi-threaded and the event-driven architectures. We use the Java platform to implement an event-driven web server with the NIO library and compare its performance with the well known Apache HTTPD web server. The goal of this study is to validate the suitability of the Java platform as a high performance web server, and to gain insight on the performance characteristics of both web server architectures under different environments such as mono-processor and multiprocessor systems. Then we analyze the impact of using SSL connections on dynamic web applications' performance, showing the impact of different SSL parameters and the benefits of running secure application servers on multiprocessor systems. We also study the scalability of servers that run secure dynamic web applications using a fine-grained analysis framework that considers all levels in the application server execution stack. Finally, the complexity of finding web server configuration that perform optimally for a specific workload in the multi-threaded web server architecture is discussed.

3.1 Web Server Architectures and Runtime Platforms

3.1.1 Introduction

The most widely-used strategies used to construct web servers are the multi-threaded architecture and the event-driven architecture. The Java platform is commonly used in web environments but at present it does not provide any standard API to implement event-driven architectures efficiently. The new 1.4 release of the J2SE introduces the NIO (New I/O) API to help in the development of event-driven I/O intensive applications. In this section we evaluate the scalability that this API provides to the Java platform in the field of web servers, comparing the well known multi-threaded Apache Httpd and an experimental server developed using the NIO API. We study the scalability of the NIO-based server as well as of its rival in a number of different scenarios, including uniprocessor, multiprocessor, bandwidth-bounded and CPU-bounded environments. **The study concludes that the NIO API can be successfully used to create event-driven Java servers which are capable of scaling as efficiently as the best of the commercial native-compiled web server, using only a few worker threads**.

The development of Web Servers is always a challenging task because it implies the creation of high performance I/O strategies. Servers attending requests from thousands of clients simultaneously need to perform really efficiently if they want to offer a good Quality of Service. Two major strategies are commonly used to confront the problem



(a) NIO **(b)** httpd2

Figure 3.1 Throughput comparison on a uniprocessor (UP) system

of servicing requests in a Web Server. The first approach is based on a multi-thread or multiprocess strategy, assigning a different thread or process to the service of each incoming request. The second strategy, referred to as event-driven model, is based on dividing of the request servicing process into a set of stages. One or more threads are in charge of each stage and make use of non-blocking I/O operations to respond simultaneously to a number of requests coming from different clients. The multi-threaded architecture is the most widely-used approach to dealing with concurrent clients on current web server systems.

The Java platform has been traditionally considered to be a low-performance environment with witch to develop I/O intensive applications. Even in J2EE[79] environments, Java is only used as far as it is required by the standard, and in general the entry point to a J2EE server, which is usually a Web Server, is delegated to native compiled applications. This is the case of many commercial J2EE application servers, such as the WebSphere Application Server[53] from IBM, which choose the Apache [81] web server as its web entry point server. But this situation could change if the new API for efficient I/O operations included in the J2SE platform since its 1.4 release really offers a high-performance I/O infrastructure. This new API, called NIO[80] (New I/O), provides the Java platform with a number of features traditionally available in most native compiled environments but still lacking in the J2SE APIs. These features include the memory mapping of files and the availability of non-blocking I/O operations.

In this section we evaluate the scalability of an experimental event-driven Java web server in a number of scenarios, bringing it together with the widely used Apache HTTPD web server in our testing platform to compare their scalability on uniprocessor,

multiprocessor, bandwidth-bounded and CPU-bounded environments. The obtained results demonstrate that the NIO API can be used to create Java web servers scaling (with the load and with the number of available processors) as well as the best of the native-compiled commercial servers with a fraction of the resources, independently of the execution scenario.

The use of event-driven architectures for web servers is an well-explored area. Flash[67] is an asymmetric multi-process event-driven web server which exploits the creation of a set of helper processes to avoid thread-blocking in the main servicing process. Haboob[85] is also an event-driven web server, based on the concepts of staged event-driven servers introduced by SEDA[85]. JAWS[52] web server uses the Proactor[51] pattern to easily construct an event-driven concurrency mechanism. In [86], the authors propose a design framework to construct concurrent systems based on the combination of threading and event-driven mechanisms to achieve both high throughput and good parallelism. Some performance issues involving event-driven architectures have been studied in [60], [92] and [49]. The introduction of a non-blocking I/O API in the J2SE was preceded by the development of a popular API called NBIO[84] (Non-Blocking I/O), which was used to create the standard API NIO. None of the previously mentioned articles evaluate scalability and performance, with respect to workload intensity and the number of processors, of the event-driven architectures for Java web servers in comparison to the more commonly used multi-threaded web servers.

The remainder of theis section is organized as follows: Section 3.1.2 discusses some important aspects of web server design, and presents some of the new features introduced in the Java platform with the Java NIO API. Sections 3.1.3 and 3.1.4 deal with the execution environment and the application benchmark used for these experiments, respectively. Section 3.1.5 evaluates the performance of the web server examined in this chapter in mono-processor environments. Finally, section 3.1.6 evaluates the NIO web server and the Apache 2 web server in multiprocessors environments.

3.1.2 Multithreaded and Event-driven architectures

The action of accepting new incoming requests on the server is usually performed by a thread which is always listening at a concrete port. When a new request arrives to the server, the request is read and then the server can behave in one of two ways depending on the under-laying I/O capabilities available in the execution environment: it either uses one thread, performing blocking I/O operations for each active socket, or it uses one thread performing non-blocking I/O operation among all the active sockets. What really distinguishes these two options is the fact that in the first case the thread which performs

the blocking I/O operation is no longer available for processing new incoming requests until the service is completed, while in the second case the thread can continue attending other clients through other sockets. This difference of concept results in a completely different way of implementing servers: if the first version is chosen, a number of threads must be created in the server in order to respond to a high number of clients. If the second version is chosen, just one thread could, theoretically, attend a high number of clients simultaneously. Once a request has been read from the client socket, the web server has to provide the client with the data requested through the connection socket.

The NIO package

The 1.4 release of the Java 2 Standard Edition includes a new set of I/O capabilities created to improve the performance and scalability of intense I/O applications. The classic Java I/O mechanisms do not offer the desired level of efficiency to run applications with high I/O requirements. Some widely existing I/O capabilities implemented on most operating systems such as mapping files to memory, readiness selection of channels and non-blocking I/O operations, were not supported by previous versions of the J2SE and have been included in the 1.4 release. The NIO package includes a readiness selection operation on channels (SocketChannels and FileChannels). This select operation is used in the implementation of the main loop to detect which web clients have sent requests to the server or which sockets have more data available. The non-blocking features of the NIO API have been used to create a web server core, based on the event-driven strategy, which is used in this chapter to evaluate the capabilities of a server based on this technology. This experimental web server will be referred to from now as the NIO server.

3.1.3 Testing environment

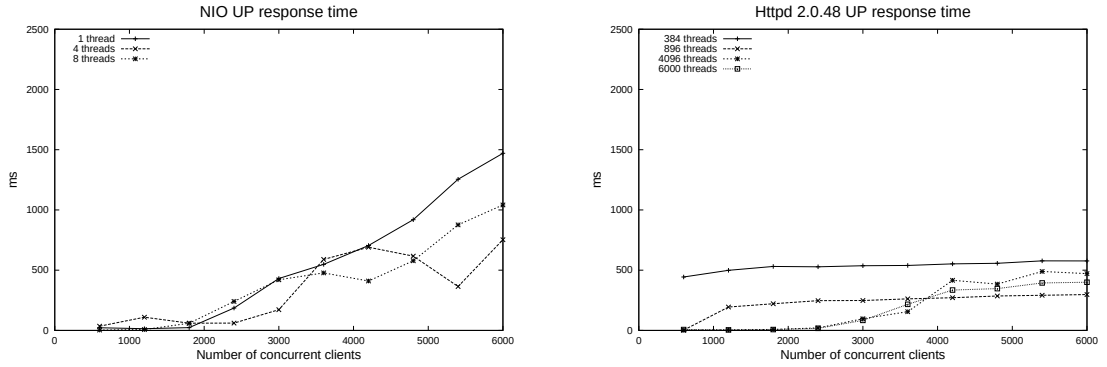
To perform the experiments discussed in this chapter, a 4-way 1.4Ghz Xeon machine with 2GB of memory was used to run the servers, and a two 2-way 2.4Ghz Xeon system with 2GB of memory was used to run the workload generators. All machines were running Linux as the operating system with a 2.6.2 kernel. The Java Runtime Environment (JRE) 1.4.1 from IBM was the chosen J2SE platform to run the Java servers. The commercial server chosen for the comparison was Apache 2.0.48, which was configured using a multi-thread schema instead of a multiprocess strategy.

We used three different configurations for the network connection between the System under Test (SUT) and the workload generators. The first configuration connected the

server machine to one client machine through a 100 Mbits Ethernet interface. The second configuration connected each one of the two client machines to the SUT through a 100 Mbit/s Ethernet interface, so we obtained an accumulated bandwidth between the server and the clients of 200 Mbit/s. Finally, the third configuration connected the SUT with a client machine through a gigabit ethernet interface. Each connection between each client machine and the SUT used a crossed-link wire to avoid collisions.

3.1.4 Application benchmark

The benchmark used to generate the workload for the experiments was Httperf, described in Section 2.4.1. Briefly, this workload generator and web performance measurement tool allows for the creation of a continuous flow of HTTP requests issued from one or more client machines and processed by one Web server machine: the SUT (System Under Test). The configuration parameters of the benchmarking tool used for the experiments presented in this chapter were set to create a realistic workload, with non-uniform reply sizes, and to sustain a continuous load on the web server. One of the parameters of the tool represents the number of emulated clients on the client machine. Each emulated client issues a number of requests, some of them pipelined, over a persistent HTTP connection.



(a) NIO **(b)** httpd2

Figure 3.2 Response time comparison on a uniprocessor (UP) system

The workload distribution generated by the Httperf benchmark was extracted from the Surge workload generator described in Section 2.4.2. The distributions produced by Surge are based on the observation of some real web server logs, here it was extracted a data distribution model of the observed workload. This fact guaranties that the used workload for the experiments follows a realistic load distribution.

3.1.5 Scalability on uniprocessors

The objective of this experiment was to compare the behavior of the NIO web server to the Apache 2 server. As a previous sub-objective of this task, we had to determine which configuration for each server, experimental or commercial, offered the best results so that we could compare the best configuration of each server. For this experiment the SMP support for the kernel of the SUT was disabled, and thus system ran as an uniprocessor environment.

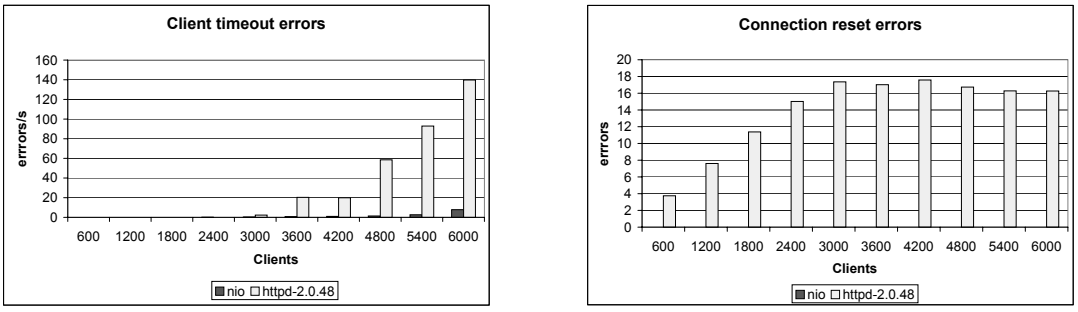
Configuration effects

For this test, each server, the NIO server and the httpd2 (Apache 2.0.48), was tested under a number of settings, in order to establish which one produced the best performance for each Web Server. The NIO server was tested with 1, 4 and 8 worker threads and the httpd2 server was configured with 500, 1000, 4000 and 6000 threads. Each test was run for a time period of 5 minutes, with the Httperf benchmark configured to produce a constant workload intensity equivalent to having a range of 600 to 6000 concurrent clients, and each connected client producing an average of 6,5 requests grouped in a session. For this first experiment, the workload was entirely produced on one client machine connected to the SUT through the gigabit Ethernet link, which ensured that the test was not bandwidth-bounded.

The obtained throughput for each server can be seen in Figure 3.1. The response time for the NIO and httpd2 servers can be seen in Figure 3.2. The throughput results in Figure 3.1 suggest that the httpd2 scales better than the NIO server when the load increases. Its throughput raises linearly with respect to the intensity of the generated workload, whereas the results obtained for the NIO server indicate that on a uniprocessor system it has more problems adapting the increasing workload intensity.

With regard to the configurations of the servers, it is noteworthly that, in the case of the httpd2 server the size for the thread pool which offered the best result for this experiment was 4096. According to our experiments, configuring the httpd2 server with a pool size of 6000 threads offered a slight performance increase with respect to the 4096 threads configuration, but significantly reduced the reliability of the system, even to the extent of hanging the system several times. As to the NIO configurations, it is specially noticeable that with only one or two worker threads (with an additional acceptor thread) it achieved the same throughput as the httpd2 server with 4096 threads. This characteristic of the NIO server is specially important on a Java server, which is commonly limited to spawn a maximum of 1000 threads in the JVM version 1.4 and previous.

For the test, we ensured that the network bandwidth was not a limiting factor because



(a) Client timeout **(b)** Connection reset

Figure 3.3 Connection errors

the observed bandwidth usage was always under 40MB/s (results on bandwidth usage can be found in [21]), which is less than the maximum achievable bandwidth for a 1 Gbit link, which was the used connection between the SUT and the client. The client is not the bounding factor on this test either, because, as will be seen in subsection 3.1.6, higher throughput is obtained when running the same tests with the SMP support activated in the kernel. As was expected, there is a linear relation between the achieved throughput of each server and the bandwidth required by each test.

Considerations of connection errors, throughput and response time

The average response time observed for the httpd2 server, as can be seen in Figure 3.2, was surprisingly low in comparison with the results obtained for the NIO server for all configurations. As this result seemed suspicious, we studied the number of connection errors detected on the benchmark client. At first glance it seemed that, in the case of the httpd2 server, a higher number of connection errors were detected on the client machine. These errors can be divided into client timeouts and connection resets.

A client timeout error occurs when the socket timeout defined by the client expires. This timeout value is used when establishing a TCP connection, when sending a request, when waiting for a reply, and when receiving a reply. If during any of those activities a request fails to make forward progress within the allowed time, httperf considers the request to have died, closes the associated connection or session and increases the client-timeout error count. For our tests, the emulated clients established a 10 seconds socket timeout value. A connection reset error takes place when a server socket timeout expires, and it is seen from the client side of the connection as an unexpected socket close. Typically, this type of error is detected when the client attempts to send data to the server

at a time the server has already closed its end of the connection.

The results for both client timeout and connection reset errors observed in the NIO and httpd2 experiments can be seen in Figure 3.3. They are expressed in number of errors per second. An interesting conclusion about connection reset errors can be extracted from Figure 3.3b in which it can be seen that, while the NIO server never produces connection reset errors, the number of this kind of error observed for the httpd2 server is not negligible and in a phase of the test even increases linearly with respect to the workload intensity. The cause of this difference is that since the multi-threaded architecture of httpd2 binds one client connection to one server thread, it needs to free threads from their assigned clients in order to be able to process new client connections. This forces the server to automatically disconnect clients after an inactivity period. The inactivity time before closing a connection is the server socket timeout and is usually set to low values, depending on the expected workload intensity. For our experiments, we set up htppd2 with a timeout value of 15 seconds. Each client emulated by the workload generator, Httperf, produced a workload based on the alternation of activity periods and think time periods. When the think time of a client exceeds the server timeout, a connection reset error takes place. In cases where the NIO server does not associate connected clients with server threads, it does not need to apply disconnection policies to its clients. As we will see in 4 this is a key advantage of the event-driven architecture, specially when connection persistence and security issues are taken into account.

With respect to the client timeout errors, it can be observed that the number of errors of this type present in the tests for the httpd2 server is much higher than for the NIO server. The explanation for this resides in the way clients are attended. In a multi-threaded server, such as httpd2, each thread is binded to a client connection. When a request is received, the requested resource is located and a blocking I/O operation on the client socket is performed. This operation is not finished until all the response is sent to the client. This virtually sequences the way in which responses are sent to clients and means that waiting clients see their timeouts expire. On the other hand, in an event-driven server, such as NIO, one or more worker threads attend a number of clients but never perform blocking operations on the socket, and this results in a fairer way of sharing the network resources between clients. Individual responses are not sent as fast as with a multi-threaded server (i.e. they obtain higher response times) but socket inactivities are less usual, avoiding timeout expiry. When a socket send buffer is full, the current operation gets blocked and the NIO server starts attending another connected client which has room in its socket buffer for the data to be sent. This effect can be observed in Figures 3.2b and 3.2a.

Another conclusion that can be drawn from the experiments is that the event-driven

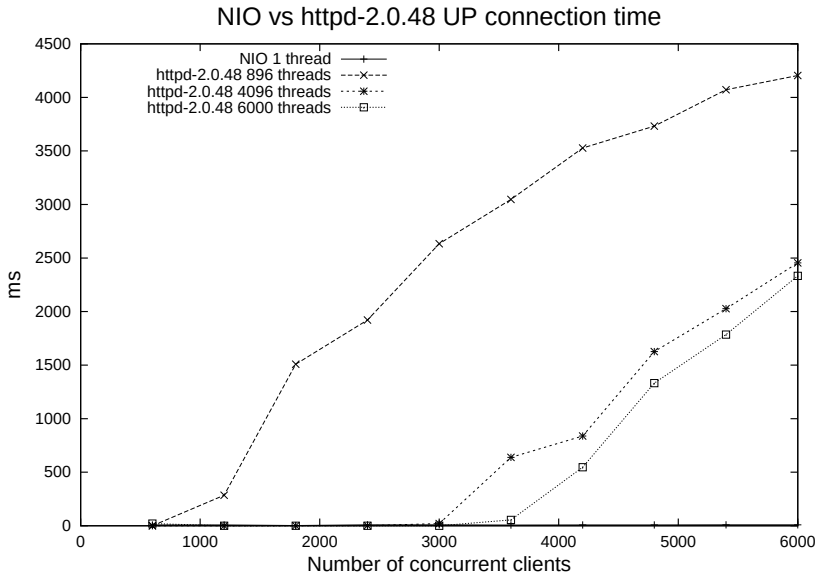


Figure 3.4 Connection time for the best configuration of NIO and httpd2

architecture makes it possible to reach shorter waiting times for each connection to be established, as can be seen in Figure 3.4. This situation is explained by the fact that the multi-threaded schemes use one thread for each client, and, as a result, when the number of simultaneous clients exceeds the number of available threads for the server, contention to access the web server appears, resulting in a sharp increase in the time needed to establish connection with the server. In the case of an event-driven server, the number of concurrent clients has nothing to do with the number of worker threads running on it and all clients can be attended without causing contention. The connection time value for the NIO server in our experiments was always below 10 ms. It is also the case that for the httpd server the point at which the connection time starts growing exponentially is not always directly related to the size of the pool of threads. When the server is set up with a pool size of 896 threads, the connection time starts degrading when the workload intensity exceeds the maximum number of available concurrent connections for the server (i.e. the number of threads). On the other hand, when the overload introduced in the system caused by the costs of managing higher pool sizes (4096 or 6000 threads) the connection time starts degrading before reaching the maximum number of available concurrent connections for the server. The high load caused by the workload intensity and the overhead caused by the management of a high number of threads present in the system is what will finally limit the maximum achievable throughput for a multi-threaded

server.

Performance effects of limiting factors

The purpose of this experiment was to study the two servers in two different conditions: when the system was bandwidth-bounded, and when it was CPU-bounded. As was pointed out in Subsection 3.1.5, the SUT kernel was configured without SMP support. For the client machines, three configurations were used: one client with a 100Mbit/s link, two clients with a 100Mbit/s link each, and one client with a one Gbit/s link.

The throughput observed in this experiment for the three network configurations can be seen in Figure 3.5, and the resulting response times are shown in Figure 3.6. When the most limiting factor in the system is bandwidth availability (in the 100 and 200 Mbit/s configurations, as is presented in [21]), both httpd2 and NIO servers have similar performance. Their throughput scales linearly with the workload intensity up to the moment when the maximum available bandwidth is reached. At that point, NIO advances the httpd2 server. This difference is hardly appreciable with a 100 Mbit/s bandwidth, and more obvious when it is increased to 200 Mbit/s. The reason for the better performance of the NIO server with respect to the httpd2 server is that the httpd2 server is obliged to apply a socket activity timeout to its clients to sustain an acceptable quality of service, and it is translated to the Ethernet level as an increase in the experienced network congestion. As is shown in Subsection 3.1.5, the number of connection reset errors is considerable for the httpd2 server and non-existent for the NIO server. This additional network load for the httpd2 server is more obvious with higher limiting bandwidths, and loses relevance when the computing capacity of the system becomes the limiting factor for the web server. On the other hand, when the bottleneck is the computing capacity of the system, the httpd2 server scales better with the workload intensity up to the moment when the overhead of rejecting a huge number of connections per second starts degrading its performance, at which point the NIO server advances it with a higher observed throughput.

Figure 3.6 shows that when the limiting factor in the system is the bandwidth, the response time for both servers is very close because, it is determined by the network capacity. When the bottleneck is the CPU (in the one Gbit/s connection), the response time observed for the servers is clearly different. The NIO server response time, as explained in Subsection 3.1.5, increases with the workload intensity because thanks to its event-driven architecture all connected clients are attended concurrently without allowing the client connection timeouts to expire. This forces the server to dedicate a part of the bandwidth to each client, thus enlarging the response time for them all. The httpd2 server responds to one client at a time, seen from the network viewpoint, and it reduces the response

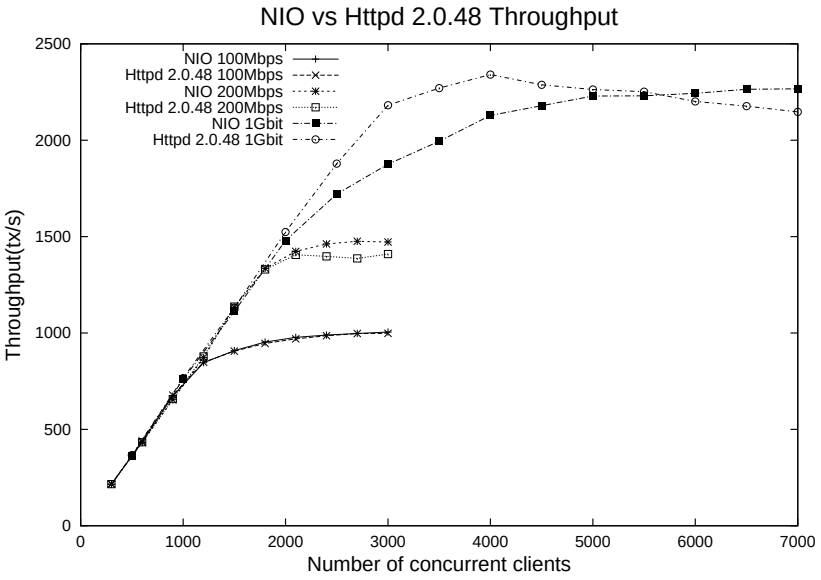


Figure 3.5 Throughput scalability on a uniprocessor (UP)

time for successful connections (those whose timeout does not expire). When the Httperf benchmark only takes into account successful connections, the observed average response time for the server is kept low.

3.1.6 Scalability on multiprocessors

Once the scalability for each web server under different workload intensities was determined with the experiments presented in Subsection 3.1.5, we explored how well each server could scale with respect to the number of available processors in the system. The procedure for the experiment was analogous to that followed in Subsection 3.1.5. First of all, some different configurations for each tested web server were studied on the SUT, now configured with full SMP support (4 processors available on the system). After that, the highest performing-configuration observed for httpd2 web server and the highest performing-configuration of the NIO-server were brought together on a SMP environment in order to compare their scalability properties.

Best-performing configurations for a SMP system

For this test, each server was tested under a number of settings to evaluate their performance trends on a multiprocessor environment. The NIO server was tested with

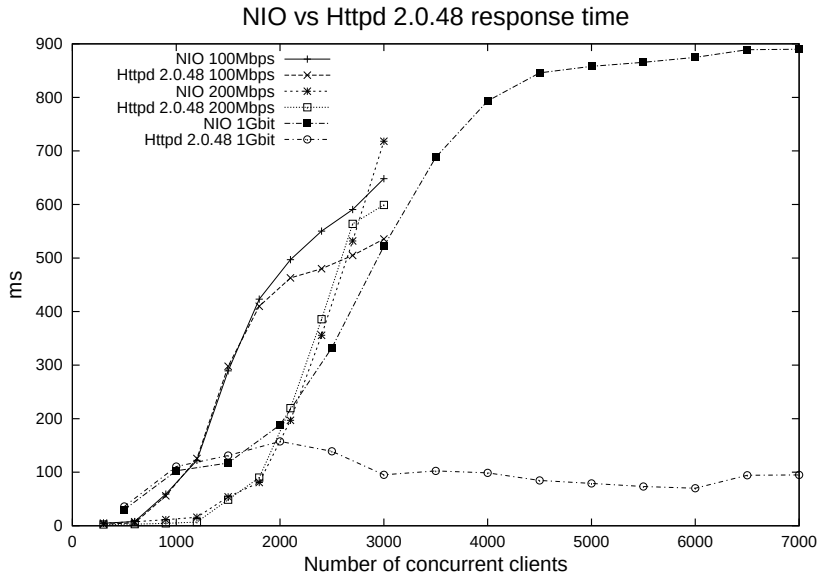
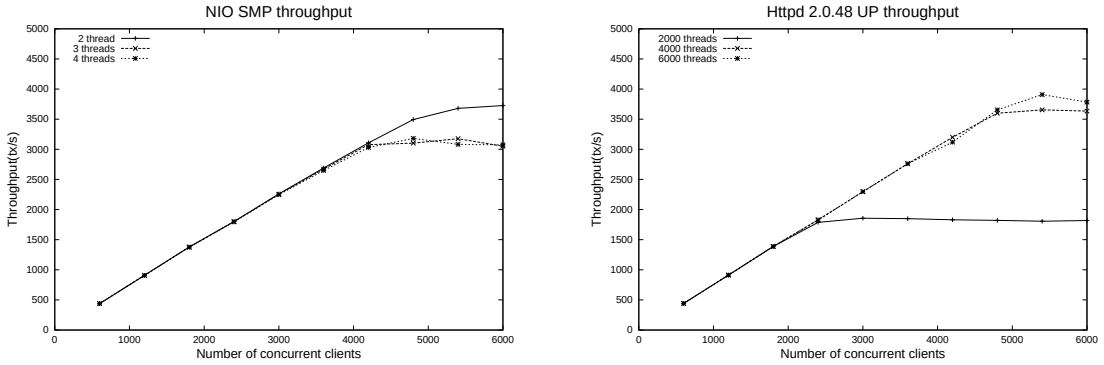


Figure 3.6 Response time scalability on a uniprocessor (UP)

2, 3 and 4 worker threads and the httpd2 server was configured with 2000, 4000 and 6000 threads. Each test was run for a time period of 5 minutes, with the Httperf benchmark configured to produce a constant workload intensity equivalent to having a range of connected clients from 600 to 6000, and each client producing an average of 6,5 requests grouped in a session. The workload was entirely produced on one client machine connected to the SUT through the 1 Gbit/s link. This ensured that the test was not bandwidth-bounded (more results on bandwidth usage can be found in [21]).

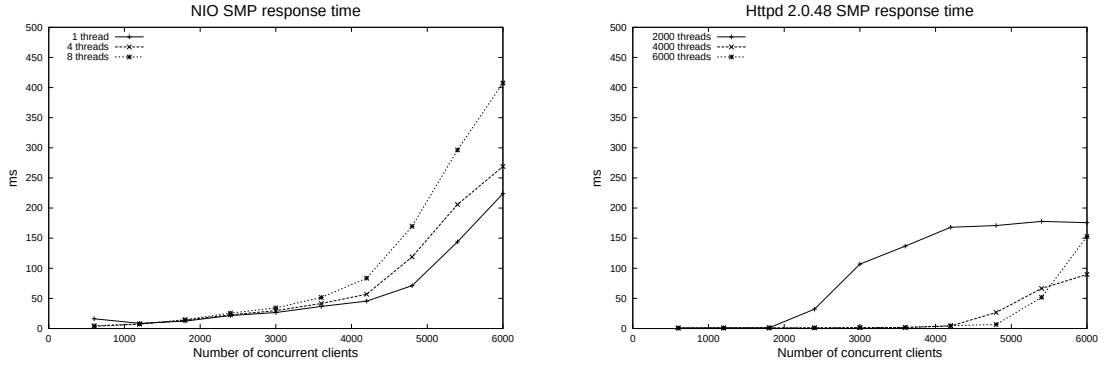
The obtained throughput for each server can be seen in Figure 3.7. The response time for the NIO and httpd2 servers can be seen in Figure 3.8. The results for the NIO and httpd2 servers' throughput, show that their performance is very similar. The httpd2 server gets higher throughput than NIO, but the difference is quite small.

The response time observed for the NIO server is worse than the results obtained for the httpd2 server, but this could again be explained by the connection errors introduced by httpd2, as has been explained in Subsection 3.1.5. The obtained values for both NIO and httpd2 are shown in Figure 3.8. The best configuration for the NIO server is that involving two worker threads. For a uniprocessor kernel, the best results were obtained with one simple worker thread, which indicates that the availability of more processors on the system favors the use of more worker threads attending clients in parallel.



(a) NIO **(b)** httpd2

Figure 3.7 Throughput comparison on a 4-way SMP system

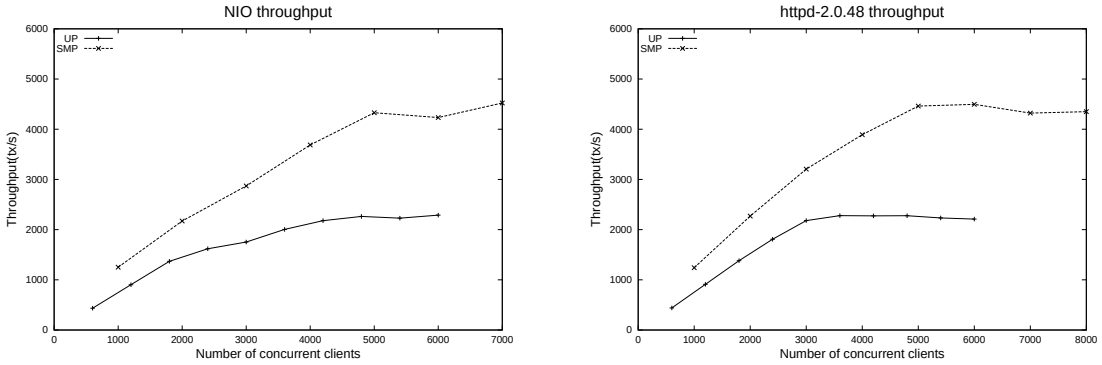


(a) NIO **(b)** httpd2

Figure 3.8 Response time comparison on a 4-way SMP system

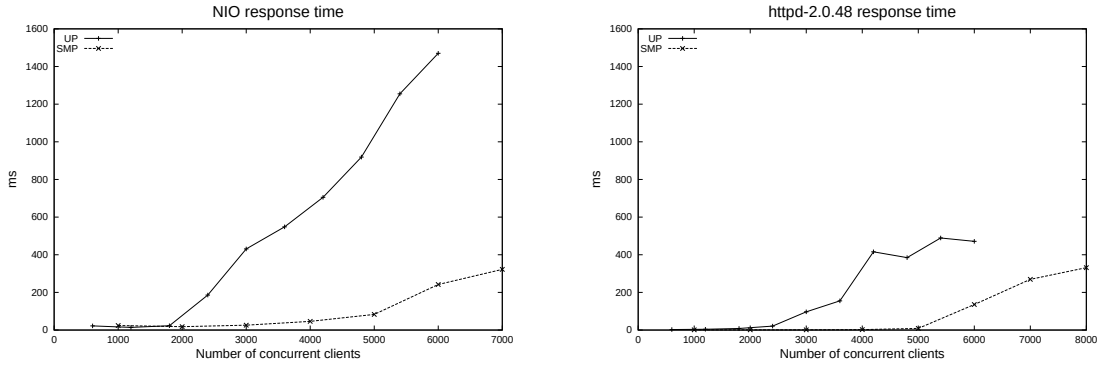
Scaling with the number of processors

The objective of the following experiment was to explore the scalability properties of the two servers when the execution environment is moved from an uniprocessor system to a multiprocessor. For the tests, we used one client machine connected to the SUT through the gigabit ethernet link. For 1-CPU configurations, a kernel without SMP support was used. For the SMP experiments, a kernel supporting the 4-way SMP system was used, so that 4 processors were available in the system. The results for the uniprocessor (UP) system are obtained from the experiments presented in Subsection 3.1.5. The server' settings for the uniprocessor and multiprocessor tests were extracted from results outlined in Subsection 3.1.5 and 3.1.6 respectively. Thus, the best settings for each environment



(a) NIO throughput **(b)** httpd-2.0.48 throughput

Figure 3.9 Throughput scalability from 1 to 4 CPUs



(a) NIO Response time **(b)** httpd-2.0.48 response time

Figure 3.10 Response time scalability from 1 to 4 CPUs

were used, namely in 2 worker threads for the NIO server and 4096 threads for the http2 server.

The results obtained reveal that the NIO server scales with the number of processors as well as the native-compiled and multi-threaded httpd2 server for the response time, shown in Figure 3.10, and for the throughput, shown in Figure 3.9. The throughput obtained by both servers on the SMP environment doubles the value obtained on the uniprocessor when it is stabilized. The values reached by the NIO and httpd2 servers are equivalent and can be considered to be in the same range of results. The observed response time for the two servers on the SMP environment is significantly lower than the obtained values for the uniprocessor configuration.

3.2 Impact of Security on Web Server Efficiency

Security in the access to web contents and the interaction with web sites is one of the most important issues affecting the performance of Internet. Servers need to provide adequate levels of security so that the user feels comfortable. HTTP over SSL (HTTP/s) is the most common protocol used in secure transactions, providing mutual authentication between both interacting parts. The SSL protocol does not introduce complexity in web applications but increases the computational demand on the server, reducing its capacity to serve large number of clients and increasing the web server response time. In order to compensate the degradation in the quality of service, the server needs to be upgraded with additional resources, mainly processors and memory.

3.2.1 Introduction

Current web sites have to face two issues that directly affect site scalability. First, the web community is growing day after day, increasing exponentially the load that sites must support in order to satisfy all client requests. Second, dynamic web content is becoming popular on current sites. Furthermore, at the same time, all information that is confidential or has market value must be carefully protected when transmitted over the open Internet. Security between network nodes over the Internet is traditionally provided using HTTPS [72]. With HTTPS, which is based on using HTTP over SSL (Secure Socket Layer [36]), it is possible to perform mutual authentication of both the sender and receiver of messages and ensure message confidentiality. This process involves the uses of X.509 certificates that can be configured on both sides of the connection. This widespread diffusion of dynamic web content and SSL increases the performance demand on application servers that host the sites. Due to these two facts, the scalability of these application servers has become a crucial issue in the bid to order to support the maximum number of concurrent clients demanding secure dynamic web content.

Characterizing application servers scalability is more complex matter than simply measuring the application server performance with different number of clients and determining the load that saturates the server. A complete characterization must also supply the causes of this saturation, giving the server administrator the opportunity and the information necessary in order to improve the server scalability by avoiding saturation. For this reason, this characterization requires powerful analysis tools that allow an in-depth analysis of the application server behavior and its interaction with the other system elements (including distributed clients, a database server, etc.). These tools must support all the levels involved in the execution of web applications (operating system, Java Virtual

Machine, application server and application) if they are to provide significant performance information to the administrators because the origin of performance problems can reside in any of these levels or in their interaction.

A complete scalability characterization must also consider another important issue: scalability relative to available resources. An analysis aimed at determining the causes of server saturation can reveal that some resource is causing a bottleneck for server scalability. In this case, a good option could be the addition of more resources of this type and the evaluation of the effect of this addition on server behavior in order to determine the causes of server saturation. On the other side, although a particular resource has been detected as a bottleneck for server scalability, the analysis of server behavior when adding more resources can be performed to verify if server saturation problem remains unresolved.

In this section we first analyze the performance obtained by using SSL connections on a multi-threaded web server, focusing on the impact of different SSL parameters and the benefits of running secure application servers on multiprocessors. Then we present a characterization of secure dynamic web applications scalability, which is divided into two parts. First, we measure the vertical scalability of the server when running with different number of processors, in order to determine the impact of adding more processors on server saturation. Second, we perform a detailed analysis of the server behavior using a performance analysis framework, in order to determine the causes of the server saturation when running with different number of processors. This framework considers all levels involved in the application server execution, allowing a fine-grain analysis of dynamic web applications.

The rest of this chapter is organized as follows: Section 3.2.2 presents the work related to performance and scalability with secure web servers. Section 3.2.3 describes in more detail the SSL protocol and how it is integrated in a web server environment. Section 3.2.4 introduces dynamic web applications. Section 3.2.5 describes our proposal for analyzing the scalability of secure dynamic web applications. Section 3.2.6 describes the experimental environment used in our evaluation. Section 3.2.7 studies the impact of SSL parameters on web server performance. Finally, Section 3.2.8 presents our evaluation of secure dynamic web applications scalability.

3.2.2 Related work

The influence of security on web server performance has been covered in various studies. For example, the performance and architectural impact of SSL on web servers in terms of various parameters such as throughput, utilization, cache sizes and cache miss

ratios has been analyzed in [11], which concluded that SSL increases computational cost of transactions by a factor of 5-7. [32] studied the impact of each individual operation of TLS protocol in the context of web servers, and showed that key exchange is the slowest operation in TLS protocol. [40] analyzed the impact of full handshake in connection establishment and proposed a caching sessions mechanism to reduce it. Boneh and Shacham [77] proposed a batch mechanism to process the complex RSA decryption that offers good speedups but introduces latency in the handshake setup. In [26] the same authors surveyed four variants of RSA designed to speed up RSA decryption and signing. Mraz [44] proposed offload RSA processing of web servers into specialized hardware by different techniques. A variety of commercial TLS hardware accelerators are inspired by this approach. Other efforts include put optimizing the TLS protocol, such as Apostolopuolos et al. [16]. As noted above, application server scalability is an important element in the provision of adequate support for the increasing number of users of secure dynamic web sites, and a great deal of effort has been made in an attempt to find the best approach to scalable application servers. To date, however, no approach, combines the application server scalability characterization when using SSL with an accurate analysis of the causes of server performance behavior. Various studies have attempted to improve application server scalability by tuning some server parameters and/or JVM options and/or operating system properties. For example, Tomcat scalability while tuning parameters that, included different JVM implementations, JVM flags and XML implementations, has been studied in [69]. Similarly, the application server scalability using different mechanisms for generating dynamic web content has been evaluated in [29]. However, none of these studies has considered any kind of scalability relative to resources (neither vertical nor horizontal), or the influence of security on the application server scalability. When considering the effect of resources addition on server scalability, most of the studies have concentrated on horizontal scalability, rather than vertical scalability, which is evaluated in this section. Major J2EE vendors such as BEA [18] or IBM [53] use clustering (horizontal scaling) to achieve scalability and high availability. Several studies have evaluated server scalability using clustering [47], while, other studies have evaluated the effect of vertical scaling on web server or application server scalability. For example, [20] and [48] only consider static web content, and the evaluation is limited to a numerical study without performing an analysis to justify the scalability results obtained. Furthermore, none of these works evaluates the effect of security on application server scalability. Certain kinds of analysis have been performed in published studies. For example, [15] and [29] provide a quantitative analysis based on general metrics of application server execution collecting system utilization statistics

(CPU, memory, network bandwidth, etc.). These statistics may allow the detection of some application server bottlenecks, but this coarse-grain analysis is often not enough to deal with more sophisticated performance problems. Our approach aims to achieve a complete characterization of secure dynamic web applications vertical scalability, determining the causes of server saturation performing a detailed analysis of application server behavior which considers all the levels involved in the execution of dynamic web applications.

3.2.3 SSL protocol

The SSL protocol provides communications privacy over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. To obtain these objectives it uses a combination of public-key and private-key cryptography algorithm and digital certificates (X.509).

The SSL protocol does not introduce a new degree of complexity in web applications structure because it works almost transparently on top of the socket layer. However, SSL greatly increases the computation time necessary to serve a connection, since it uses cryptography to achieve its objectives. This increase has a noticeable impact on server performance, which can be appreciated in Figure 3.11. This figure compares the throughput obtained by the Tomcat application server, configured as described in Section 3.2.6, using secure connections versus using normal connections. Notice that the maximum throughput obtained when using SSL connections is 72 replies/s and the server scales only until 200 clients. In contrast, when normal connections are used, the throughput is considerably higher (550 replies/s) and the server can scale up to 1700 clients. Finally, notice that when the server is saturated, if attending normal connections, the server can maintain the throughput if new clients arrive, but if attending SSL connections, the server cannot maintain the throughput and the performance is degraded.

The SSL protocol has two fundamentall phases of operation: the SSL handshake and the SSL record protocol. We will overview the SSL handshake phase, which is responsible for most of the computation time required when using SSL. A detailed description of the whole protocol can be found in RFC 2246 [36].

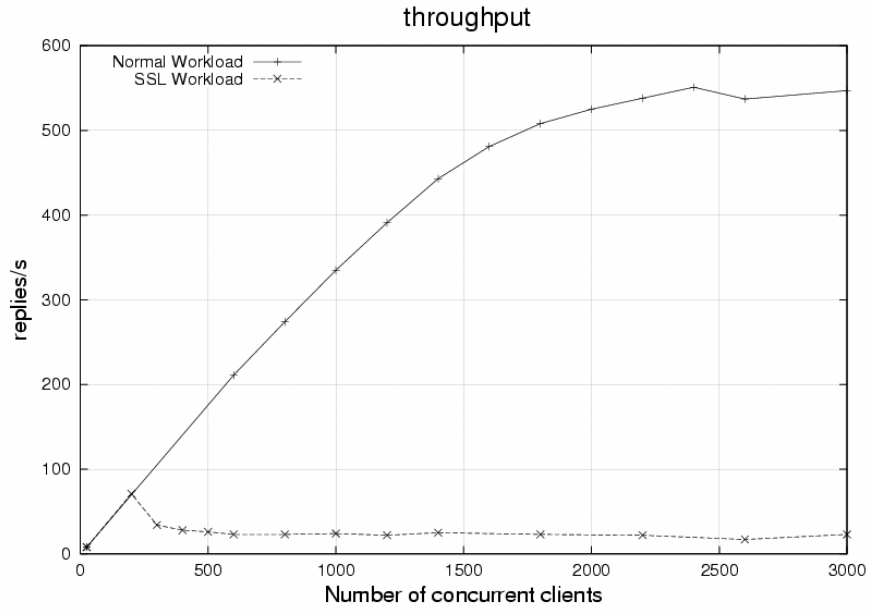


Figure 3.11 Tomcat scalability when serving secure vs. non-secure connections

SSL handshake

The SSL handshake allows the server to authenticate itself to the client using public-key techniques like RSA, and then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server. This process is detailed in Figure 3.12.

Two different SSL handshake types can be distinguished: the full SSL handshake and the resumed SSL handshake. The full SSL handshake is negotiated when a client establishes a new SSL connection with the server, and requires the complete negotiation of the SSL handshake. This negotiation includes parts that spend a lot of computation time to be accomplished. We have measured the computational demand of a full SSL handshake in a 1.4 GHz Xeon machine to be around 175 ms.

The SSL resumed handshake is negotiated when a client establishes a new HTTP connection with the server but uses an existing SSL connection. As the SSL session ID is reused, part of the SSL handshake negotiation can be avoided, which considerably reduces the computation demand for performing a resumed SSL handshake. We have measured the computational demand of a resumed handshake in a 1.4 GHz Xeon machine, and it is around 2 ms. Notice the big difference between negotiating a full SSL handshake and negotiating a resumed SSL handshake (175 ms versus 2 ms).

As we can see in Figure 3.12, the client sends a client hello message to which

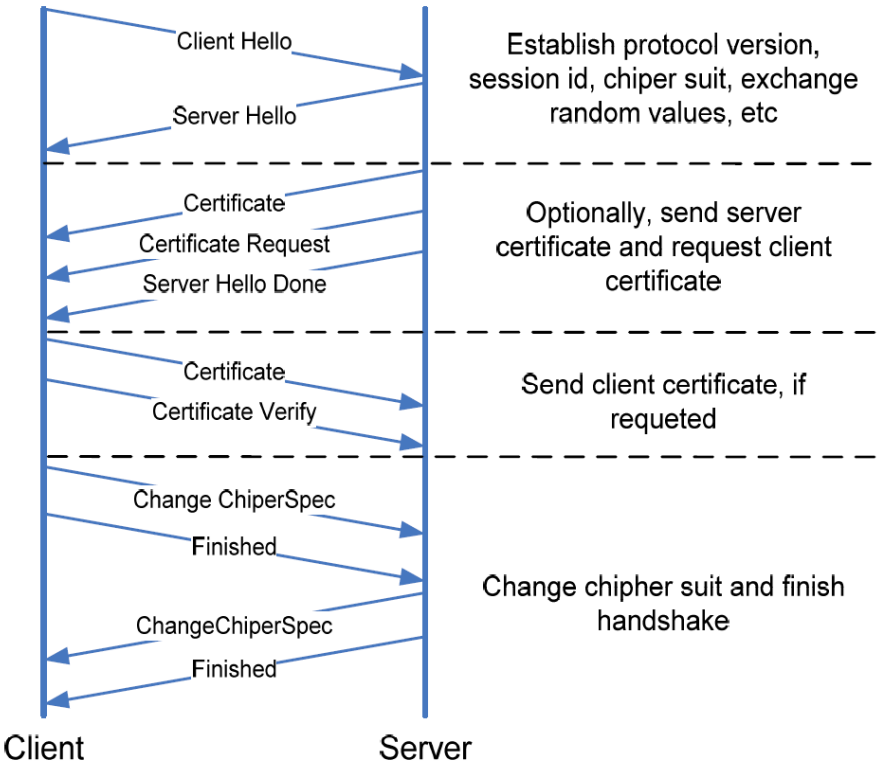


Figure 3.12 SSL Handshake protocol

the server must respond with a server hello message. The client hello and server hello establish the following attributes: protocol version, session ID, cipher suite, and compression method. Additionally, two random values are generated and exchanged. Following the hello messages, the server will send its certificate. If the server is authenticated, it may request a certificate from the client. Now the server will send the server hello done message, indicating that the hello- message phase of the handshake is complete. The client key exchange message is now sent. At this point, a change cipher spec message is sent by the client who then immediately sends the finished message. In response, the server will send its own change cipher spec message and Finished message. At this point, the handshake is complete and the client and server may begin to exchange application layer data. In the case of a resumed handshake, the client sends a client hello using the Session ID of the session to be resumed. Then the server checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a server hello with the same Session ID value. At this point, both client and server must send a change cipher spec messages and proceed directly to finished messages. Once the re-establishment is complete, the client and server may begin to exchange application layer data. If a Session ID match is not

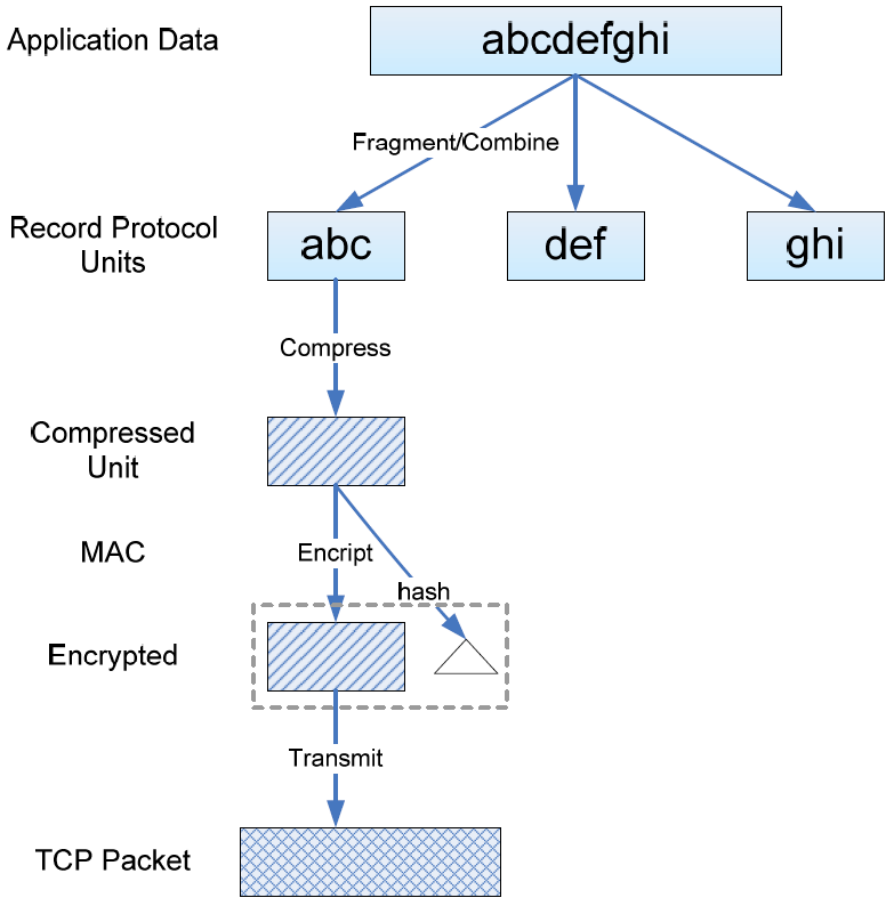


Figure 3.13 SSL Record protocol

found, the server generates new session ID and the SSL client and server perform a full handshake.

The SSL Record Protocol

The SSL Record layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size. Then the information blocks are fragmented into plain-text records of 214 bytes or less. All records are compressed using the compression algorithm defined in the current session state and protected using the encryption and MAC (Message Authentication Code) algorithms defined in the current CipherSpec. Finally encryption and MAC functions translate compressed units to encrypted data, ready to be send into TCP packet. This process is depicted in Figure 3.13.

3.2.4 Dynamic web applications

Dynamic web applications are a case of multi-tier application and are mainly composed of a Client tier and a Server tier, the latter consisting of a front-end web server, an application server and a back-end database. The client tier is responsible for interacting with application users and for generating requests to be attended by the server. The server tier implements the logic of the application and is responsible for serving user-generated requests. When the client sends an HTTP request for dynamic content to the web server, the web server forwards the request to the application server (as understood in this section, a web server only serves static content), which is the dynamic content server. The application server executes the corresponding code, which may need to access the database to generate the response. The application server formats and assembles the results into an HTML page, which is returned as an HTTP response to the client.

The implementation of the application logic in the application server may take various forms, including PHP , Microsoft Active Server Pages , Java Servlets and Enterprise Java Beans (EJB) . This study focuses on Java Servlets, but the same methodology can be applied to other mechanisms for generating dynamic web content, in order to characterize their scalability.

A servlet is a Java class used to extend the capabilities of servers which host applications accessed via a request- response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes. Servlets access the database explicitly, using the standard JDBC interface, which is supported by all major databases. Servlets can use all the features of Java. In particular, they can use Java built-in synchronization mechanisms to perform locking operations.

3.2.5 Servers scalability

The scalability of an application server is defined in terms of its ability to maintain site availability, reliability, and performance as the amount of simultaneous web traffic, or load, hitting the application server increases [48]. Given this definition, the scalability of an application server can be represented as a measurrement of the performance of an application server as the load increases. With this representation, the load that provokes the saturation of the server can be detected. We consider that the application server is saturated when it is unable to maintain the site availability, reliability, and performance (i.e. the server does not scale). This definition implies that when the server is saturated,

the performance is degraded (lower throughput and higher response time) and the number of client requests refused increase.

At this point, two questions should occur to the reader (and of course, to the application server administrator). First, the load that provokes the saturation of the server has been detected, but why does this load cause the server performance to degrade? In other words, in which parts of the system (CPU, database, network, etc.) will a request be spending most of its execution time at the saturation points? In order to answer this question, we propose to analyze application server behavior using a performance analysis framework, which considers all levels involved in the application server execution (operating system, JVM, application server and application), as this allows a fine-grain analysis of dynamic web applications. Second, the application server scalability with given resources has been measured, but how would the addition of more resources affect this application server scalability? This question adds a new dimension to application servers scalability: the measurement of the scalability relative to the resources. This scalability can be done in two different ways: vertical and horizontal.

Vertical scalability (also called scaling up) is achieved by adding capacity (memory, processors, etc.) to an existing application server and requires few or no changes to the architecture of the system. Vertical scalability increases the performance (in theory) and the manageability of the system, but decreases reliability and availability (single failure is more likely to lead to system failure). The issue of scalability relative to resources is considered later in this chapter. Horizontal scalability (also called scaling out) is achieved by adding new application servers to the system, increasing the complexity of the system. Horizontal scalability increases reliability, availability and performance (depends on load balancing), but decreases manageability, since there are more elements in the system.

The analysis of application server behavior will help us answer the question of how the addition of resources affects application server scalability. If we detect some resource which is acting as a bottleneck for application server performance, this encourages the addition of new resources of this type (vertical scaling). We can then, measure the scalability with this new configuration and analyzes the application server behavior with the performance analysis framework to determine the improvement on server scalability and the new causes of server saturation. On the other hand, if we upgrade a resource that is not causing a bottleneck for the application server performance, by using the performance analysis framework, we can verify that scalability is not improved and in this case the causes of server performance degradation remain unresolved. This observation explains why vertical scaling sometimes only improve performance in theory: it depends if the added resource is a bottleneck for server performance or not. This observation also

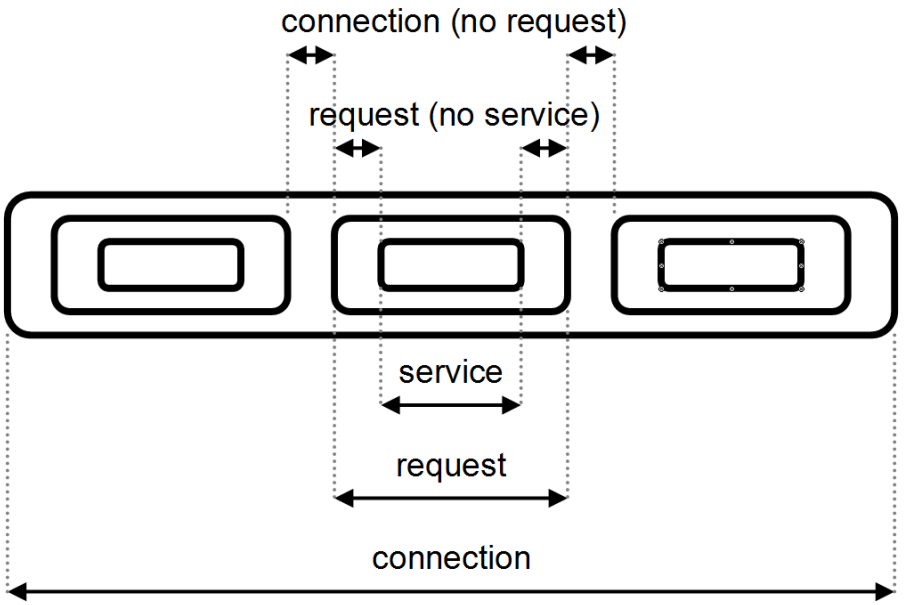


Figure 3.14 Tomcat persistent connection pattern

explains why an analysis of application server behavior, so as to detect the causes of saturation, should be carried out before new resources are added.

3.2.6 Experimental environment

Tomcat servlet container

We use Tomcat v5.0.19 [39] as the web server. Tomcat is an open-source servlet container developed under the Apache license. Its primary goal is to serve as a reference implementation of the Sun Servlet and JSP specifications, and also to be a quality production servlet container. Tomcat can work as a stand-alone server (serving both static and dynamic web content) or as a helper for a web server (serving only dynamic web content). In this thesis we use Tomcat as a stand-alone server.

Tomcat follows a connection service schema where, at any given time, one thread (an HttpProcessor) is responsible for accepting a new incoming connection on the server listening port and assigning a socket structure to it. From this point on, this HttpProcessor will be responsible for attending and serving the received requests through the persistent connection established with the client, while another HttpProcessor will continue accepting new connections. HttpProcessors are commonly chosen from a pool of threads in order to avoid thread creation overheads.

Persistent connections are a feature of HTTP 1.1 that allows it to serve different requests using the same connection, thus saving a lot of work and time for the web server,

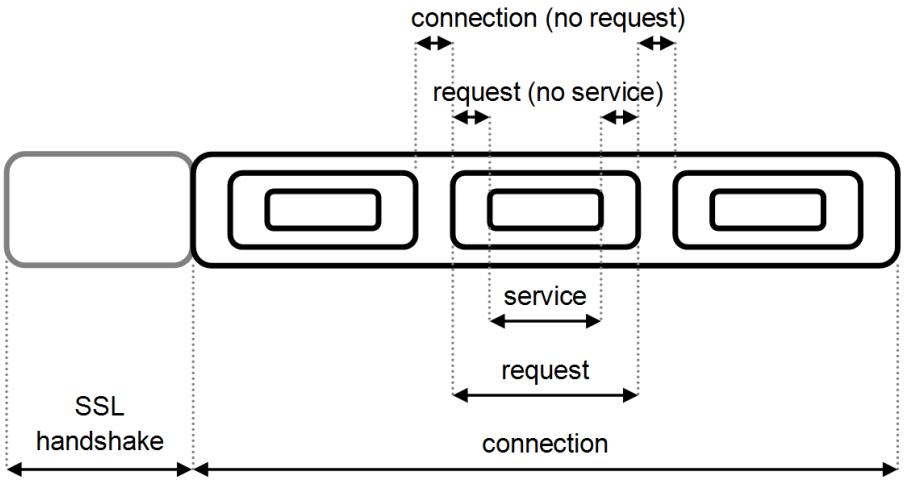


Figure 3.15 Tomcat secure persistent connection pattern

client and the network, considering that establishing and tearing down HTTP connections is an expensive operation.

The pattern of a persistent connection in Tomcat is shown in Figure 3.14. In this example, three different requests are served through the same connection. The rest of the time (connection (no request)) the server maintain the connection open, waiting for another client request. A connection timeout is programmed to close the connection if no more requests are received. Notice that within every request the service (execution of the servlet implementing the demanded request) is distinguished from the request (no service). This is the pre and post process that Tomcat requires to invoke the servlet which then implements the demanded request.

Figure 3.15 shows the pattern of a secure persistent connection in Tomcat. Notice that when using SSL the pattern of the HTTP persistent connection is maintained, but the underlying SSL connection supporting this persistent HTTP connection must be established previously, negotiating a SSL handshake, which can be full or resumed depending on whether or not a SSL Session ID is re-used. For instance, if a client has to establish a new HTTP connection because its current HTTP connection has been closed by the server due to connection persistence timeout expiration, as it re-uses the underlying SSL connection, it negotiates a resumed SSL handshake. We have configured Tomcat setting the maximum number of HttpProcessors to 100 and the connection persistence timeout to 10 seconds.

Auction site benchmark (RUBiS)

The experimental environment also includes the deployment of the RUBiS (Rice University Bidding System) [15] benchmark servlets version 1.4 on Tomcat. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. RUBiS defines 27 interactions. Among the most important ones are browsing items by category or region, bidding, buying or selling items and leaving comments on other users. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. RUBiS supplies implementations using some mechanisms for generating dynamic web content shuch as PHP, Servlets and several kinds of EJB.

The client workload for the experiments was generated using a workload generator and web performance measurement tool called Httperf 2.4.1. This tool, which support both HTTP and HTTPS protocols, allows for the creation of a continuous flow of HTTP/S requests issued from one or more client machines and processed by one server machine: the SUT (System Under Test). The configuration parameters of the benchmarking tool used for the experiments presented were set to create a realistic workload, with non-uniform reply sizes, and to sustain a continuous load on the server. One of the parameters of the tool represents the number of concurrent clients interacting with the server. Each emulated client opens a session with the server. The session remains alive for a period of time, called session time, at the end of which the connection is closed. Each session is a persistent HTTP/S connection with the server. Using this connection, the client repeatedly makes a request (the client can also pipeline some requests), parses the server response to the request, and follows a link embedded in the response. The workload distribution generated by Httperf was extracted from the RUBiS client emulator, which uses a Markov model to determine which subsequent link from the response to follow. Each emulated client waits for an amount of time, called the think time, before initiating the next interaction. This emulates the thinking period of a real client who takes a period of time before clicking on the next request. The think time is generated from a negative exponential distribution with a mean of 7 seconds. Httperf also allows client timeout to be configured. If this timeout elapses and no reply has been received from the server, the current persistent connection with the server is discarded, and a new emulated client is initiated. We have configured Httperf setting the client timeout value to 10 seconds. RUBiS defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions.

Performance analysis framework

In order to determine the causes of server saturation, we propose analyzing application server behavior using a performance analysis framework. This framework, which consists of an instrumentation tool called Java Instrumentation Suite (JIS [46]) and a visualization and analysis tool called Paraver [70],, considers all levels involved in application server execution (operating system, JVM, application server and application), allowing a fine-grain analysis of dynamic web applications. For example, the framework can provide detailed information about thread status, system calls (I/O, sockets, memory and thread management, etc.), monitors, services, connections, etc. Further information about the implementation of the performance analysis framework and its use for the analysis of dynamic web applications can be found in [29] and [12].

Hardware & software platform

Tomcat runs on a 4-way Intel XEON 1.4 GHz with 2 GB RAM. We use MySQL v4.0.18 [7] as our database server with the MM.MySQL v3.0.8 JDBC driver. MySQL runs on a 2-way Intel XEON 2.4 GHz with 2 GB RAM. We also have a 2-way Intel XEON 2.4 GHz with 2 GB RAM machine running the workload generator (Httperf 0.8). Each client emulation machine emulates the configured number of clients performing requests to the server for 10 minutes using the browsing mix (read- only interactions). All the machines run the 2.6.2 Linux kernel. The server machine is connected to the client machine through a 1 Gbps Ethernet interface. The database and server machine is directly connected through 100 Mbps fast Ethernet crossed-link. For our experiments we use the Sun JVM 1.4.2 for Linux, using the server JVM instead of the client JVM and setting the initial and maximum Java heap size to 1024 MB, which we have proved to be enough to avoid memory being a bottleneck for performance. All the tests are performed with the common RSA- 3DES-SHA cipher suit. Handshake is performed with 1024 bit RSA key. Record protocol uses triple DES to encrypt all application data. Finally, SHA digest algorithm provides the Message Authentication Code (MAC).

3.2.7 SSL protocol evaluation

In this section we evaluate the effect of different client behaviors relative to SSL on server performance. We want to show that different client patterns (Do the clients reuse the SSL session IDs?, Do the clients retry the erroneous requests? How long do the clients wait for a server response?) can seriously affect server performance. Our methodology consists of varying one parameter and comparing the throughput obtained with the one

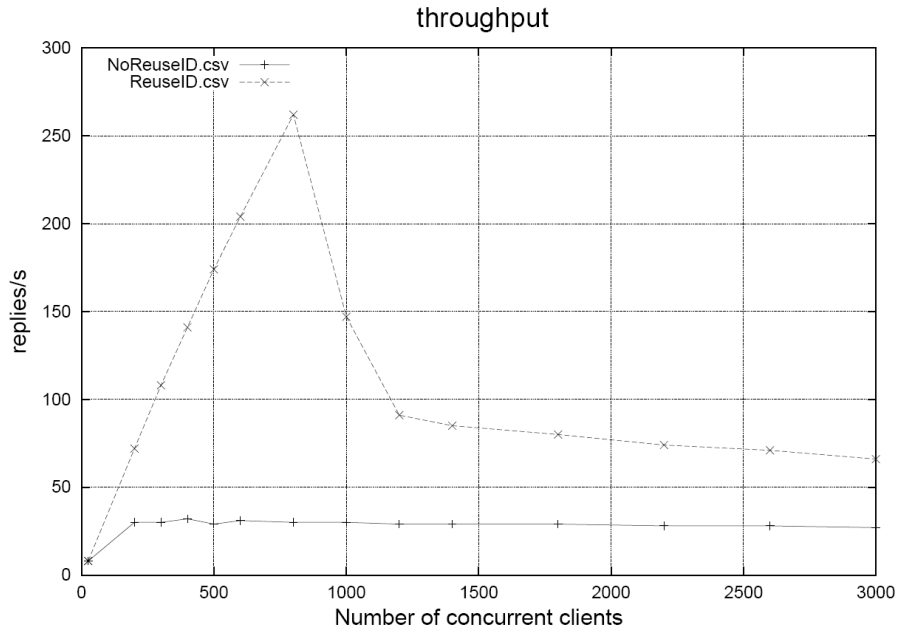


Figure 3.16 Standard configuration vs. No Reuse Session ID

obtained with the standard configuration described in the previous section above. First, we evaluate the effect of not reusing the SSL connections. Second, we will evaluate the server throughput supposing that a new client is initiated when an error is produced (instead of retrying the failed requests). Finally, we evaluate the server throughput when the number of client connections is reduced by programming the clients to wait indefinitely for the server responses.

Reuse session ID

As already stated above, SSL connections can be re-used in order to reduce the cost of the handshake phase. But Httperf can be forced to do a full handshake every time a connection is established. This has a very big effect of performance, as shown in Figure 3.16. If SSL session ID it is not re-used, each client connection with the server has to negotiate a full SSL handshake. We have explained in previous sections that there is big difference in processing demand between negotiating a full SSL handshake (around 175 ms) and negotiating a resumed SSL handshake (around 2 ms). This explains the large performance degradation showed in Figure 3.16.

Retry on failure

This parameter determines client behavior when an error is produced. It is enabled by default, with the result that if an error is detected, the same URL is requested again

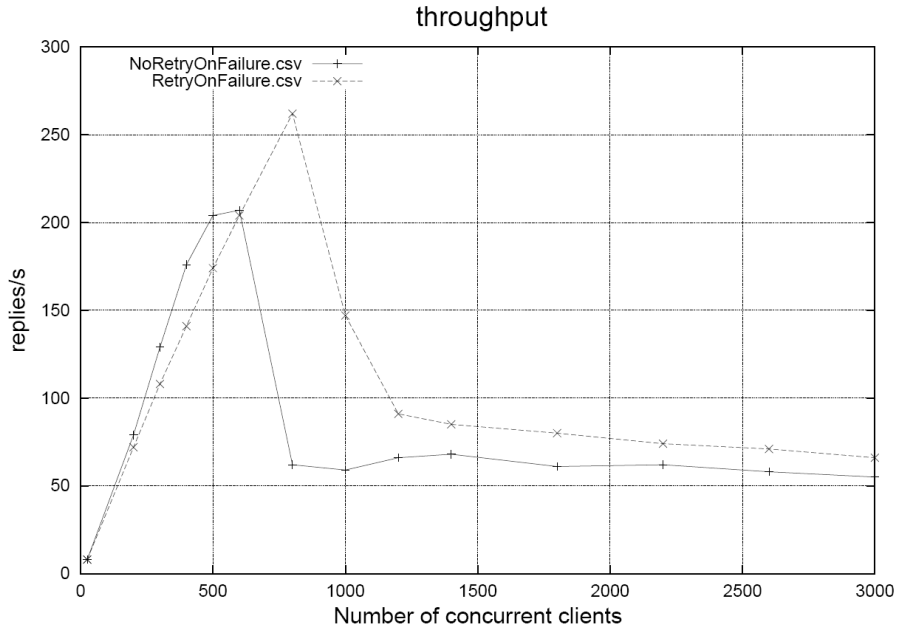


Figure 3.17 Standard configuration vs. No Retry On Failure

(this implies that the SSL session ID is re-used). On the other hand, if this parameter is disabled, the current client is aborted and a new client is initiated (provoking the establishment of a new SSL connection with the server). Figure 3.17 shows the effect on throughput when using this parameter. Notice that, the fewer SSL connections established, the better the throughput.

Client Timeout

In this section we evaluate the server behavior when the number of client re-connections is reduced due to timeouts. Notice that when using secure connections, the cost of establishing re- connections is critical for performance. In order to achieve this reduction, we parameterize the client timeout to an infinite value, and thus the clients will wait for a server response indefinitely, reducing in this way the number of re-connections to the server. Figure 3.18 shows the throughput achieved. Notice that when clients wait for server response indefinitely, throughput is maintained, while when setting the client timeout, throughput degrades considerably due to the high number of client re-connections. On the basis of these results, controlling the number of client connections that the server is able to process seems recommendable. For example, we could introduce a connection manager prepared to detect situations where a high number of connections are negatively affecting the server performance, and, at this point, decide to limit the maximum number of connection establishments per unit of time to avoid performance

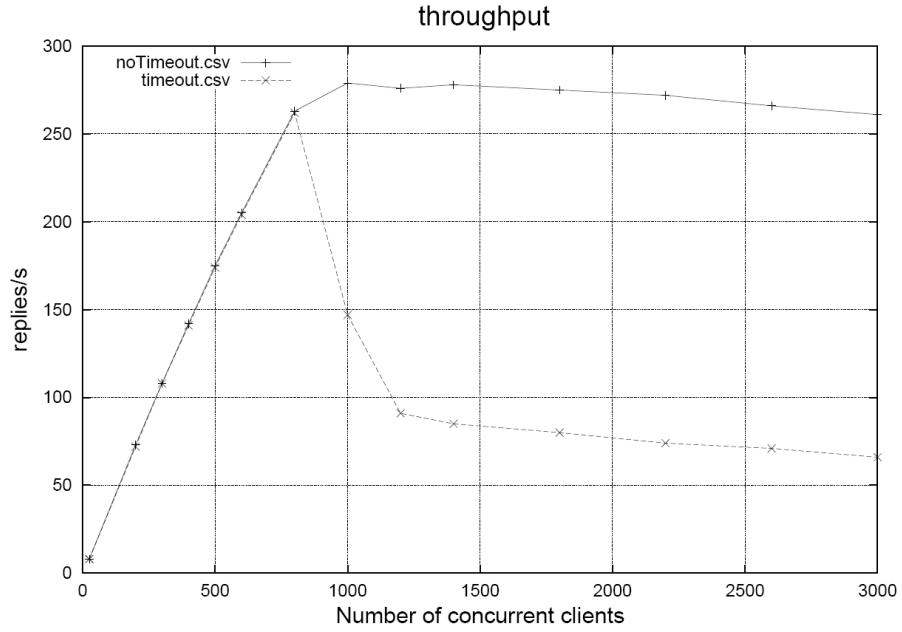


Figure 3.18 Standard configuration vs. Infinite Client Timeout

degradation.

3.2.8 Tomcat scalability evaluation

In this section we present the scalability characterization of Tomcat application server when running the RUBiS benchmark using SSL. The evaluation is divided in two parts. First, we evaluate the vertical scalability of the server when running with different number of processors, in order to determine the impact of adding more processors on server saturation (Can the server support more clients before saturating?) Second, we perform a detailed analysis of server behavior using a performance analysis framework, in order to find the causes of server saturation when running with different numbers of processors.

Tomcat vertical scalability

Table 3.1 Number of clients that saturate the server and maximum achieved throughput before saturating

number of processors	number of clients	throughput (replies/s)
1	250	90
2	500	172
4	950	279

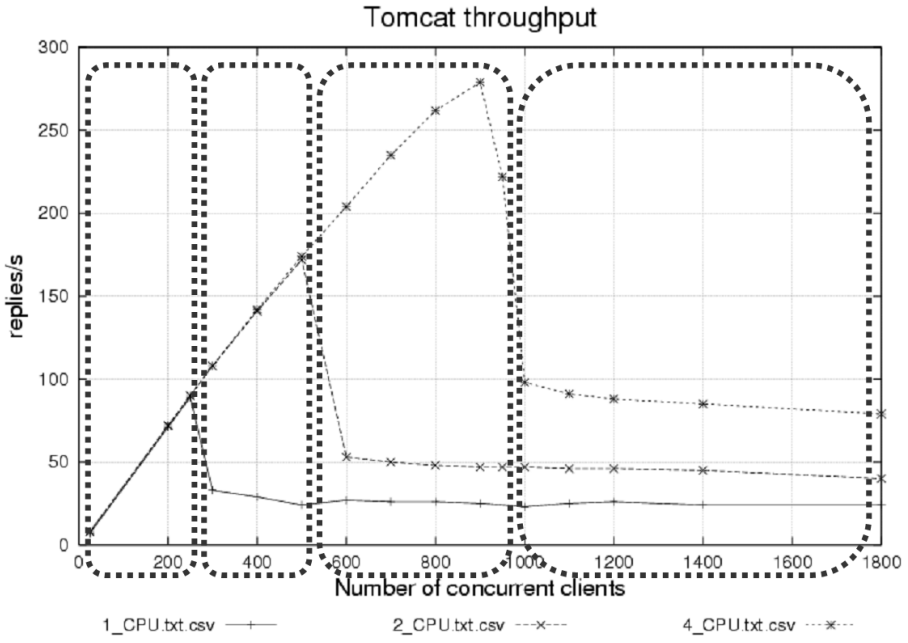


Figure 3.19 Tomcat scalability with different number of processors

Figure 3.19 shows the Tomcat scalability when running with different numbers of processors, representing the server throughput as a function of the number of clients. Notice that for a given number of processors, the server throughput increases linearly with respect to the input load (the server scales) until a determined number of clients hit the server. At this point, the throughput achieves its maximum value. Table 3.1 shows the number of clients that saturate the server and the maximum achieved throughput before saturating when running with one, two and four processors. Notice that running with more processors allows the server to handle more clients before saturating, so the maximum achieved throughput is higher.

Notice also that the same throughput can be achieved, as shown in Figure 3.11, with a single processor when SSL is not used. This means that when using secure connections, the computing capacity provided when adding more processors is spent on supporting the SSL protocol.

When the number of clients that saturate the server is reached, the server throughput degrades to approximately the 30% of the maximum achievable throughput, as shown in Table 3.2. This table shows the average throughput obtained when the server is saturated when running with one, two and four processors. Notice that, although the throughput obtained is degraded in all cases when the server reaches a saturated state, running with more processors improves the throughput (duplicating the number of processors, the throughput almost duplicates too).

Table 3.2 Average server throughput when saturated

number of processors	throughput (replies/s)
1	25
2	50
4	90

Tomcat scalability analysis

In order to perform a detailed analysis of the server, we selected four different loads: 200, 400, 800 and 1400 clients, each one corresponding to one of the zones observed in Figure 3.19. These zones group the loads with similar behavior of the server. In order to conduct this analysis, we used the performance analysis framework described in Section 3.2.6.

The analysis methodology consists of comparing server behavior when it is saturated (400 clients when running with one processor, 800 clients when running with two processors and 1400 clients when running with four processors) with when it is not (200 clients when running with one processor, 400 clients when running with two processors and 800 clients when running with four processors). We calculated a series of metrics representing server behavior, and determined which of them were affected when the number of clients was increased. From these metrics, an in-depth analysis was performed in order to find the causes of their dependence on server load.

In order to detect the causes of server saturation we used the performance analysis framework to calculate, the average time spent by the server to process a persistent client connection, distinguishing the time devoted to each phase of the connection (connection phases have been described in Section 3.2.6 when running with different numbers of processors. This information is displayed in Figure 3.20. As shown in this figure, running with more processors decreases the average time required to process a connection. Notice that when the server is saturated, the average time required to handle a connection increases considerably. Going into detail on the connection phases, the time spent on the SSL handshake phase of the connection increases from 28 ms to 1389 ms when running with one processor, from 4 ms to 2003 ms when running with two processors and from 4 ms to 857 ms when running with four processors, becoming the phase where the server spends most of the total time needed to process a connection.

To determine the causes of the large increase in the time spent on the SSL handshake phase of the connection, we calculated the percentage of connections that perform a resumed SSL handshake (re-using the SSL Session ID) and the percentage of connections that perform a full SSL handshake when running with different numbers of processors.

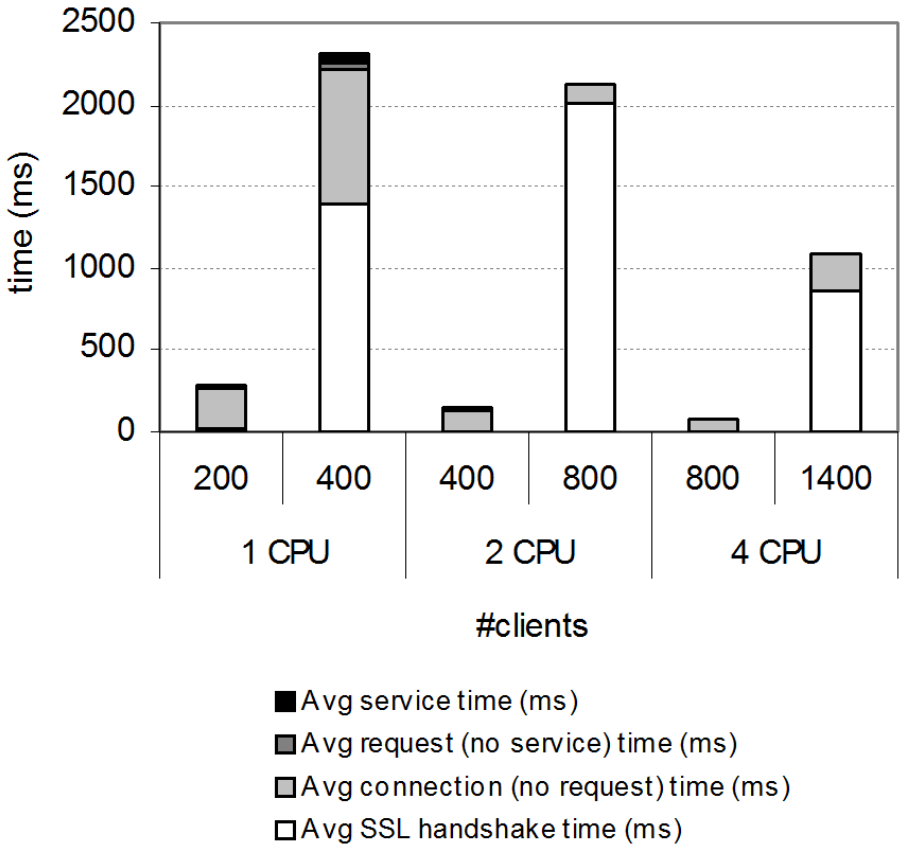


Figure 3.20 Average time spent by the server processing a persistent client connection

This information is shown in Figure 3.21. Notice that when running with one processor and with 200 clients, 97% of SSL handshakes can reuse the SSL connection, but with 400 clients, only 27% can reuse it. The rest must negotiate the full SSL handshake, saturating the server because it cannot supply the computational demand of these full SSL handshakes. Remember the big difference between the computational demand of a resumed SSL handshake (2 ms) and a full SSL handshake (175 ms). The same situation is produced when running with two processors (the percentage of full SSL handshakes has increased from 0.25% to 68%), and when running with four processors (from 0.2% to 63%).

We have determined that when running with any number of processors the server saturates when most of the incoming client connections must negotiate a full SSL handshake instead of resuming an existing SSL connection, requiring a computing capacity that the available processors are unable to supply. Nevertheless, the question remains, Why does this occur for a given number of clients? In other words, Why do incoming connections negotiate a full SSL handshake instead of a resumed SSL handshake when attending a given number of clients? Remember that we have configured

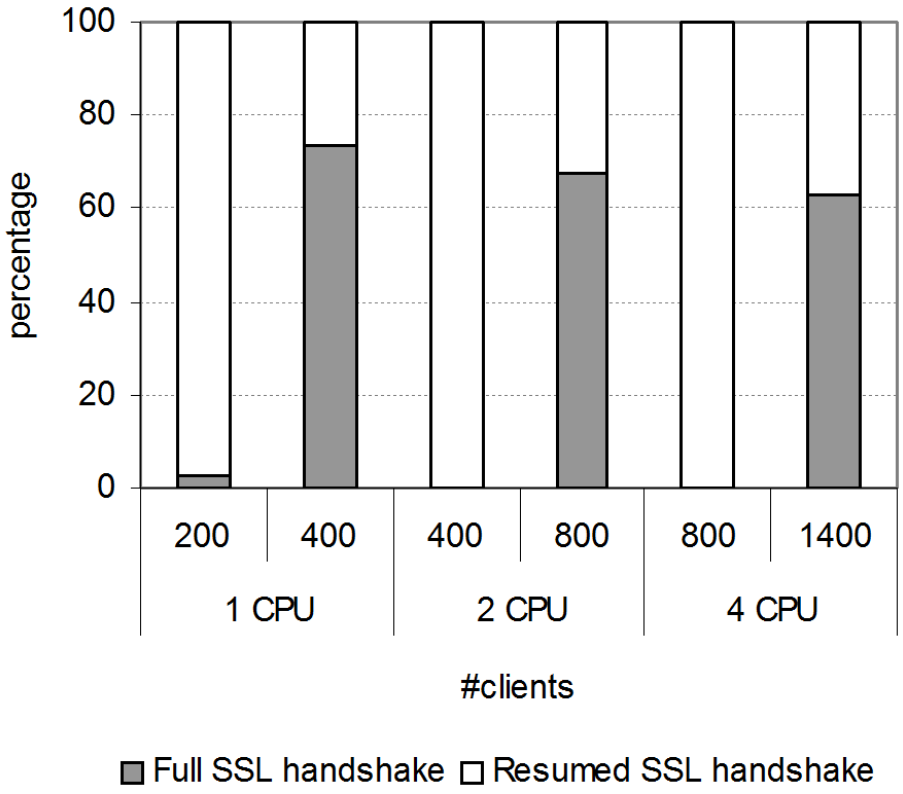


Figure 3.21 Incoming server connections classification depending on SSL handshake type performed

the client with a timeout of 10 seconds. This means that if no reply is received in this time (the server is unable to supply it because it is heavy loaded), this client is discarded and a new one is initiated. Remember, also, that the initiation of a new client requires the establishment of a new SSL connection, and therefore the negotiation of a full SSL handshake.

It follows from the above that, if the server is loaded and cannot handle the incoming requests before the client timeouts expire, avery large number of new clients arrive, and these connections need the negotiation of a full SSL handshake. This causes the server performance degradation. This assertion is supported by the information in Figure 3.22, which shows the number of client timeouts that occurred when running with different numbers of processors. Notice that for a given number of clients, the number of clients timeouts increases considerably, because the server is unable to respond to the clients before their timeouts expires. The comparison of this figure with Figure 3.19 reveals that this given number of clients matches the saturation load of the server.

In order to evaluate the effect on the server of the large number of full SSL handshakes, we calculated, using the performance analysis framework, the state of HttpProcessors

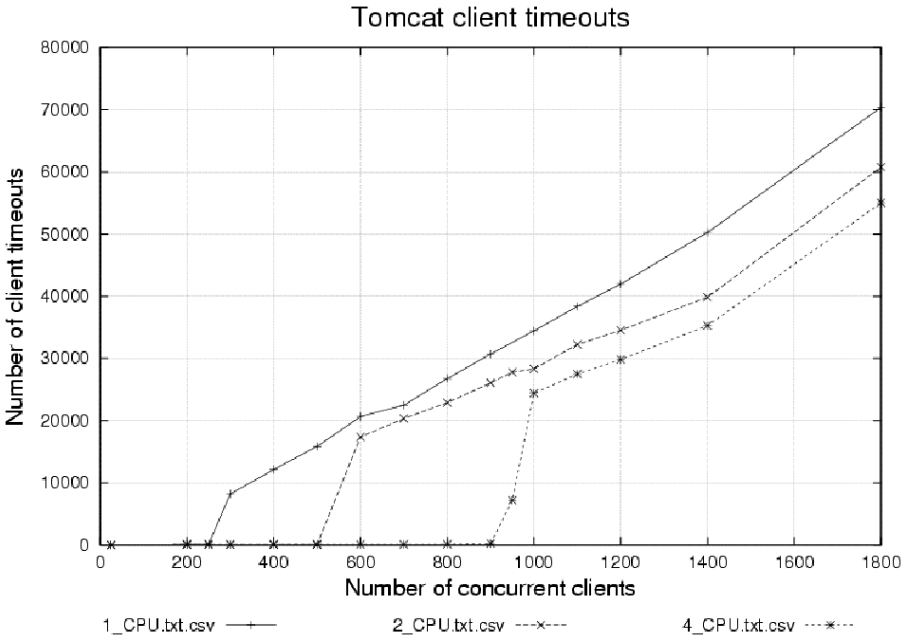


Figure 3.22 Client timeouts with different number of processors

when they are in the SSL handshake phase of the connection. This information is given, in Figure 3.23. The HttpProcessors can be running (Run state), blocked waiting for the finalization of an input/output operation (Blocked I/O state), blocked waiting for the synchronization with other HttpProcessors in a monitor (Blocked Synch) or waiting for a free processor to become available to execute (Ready state). When the server was not saturated, HttpProcessors spent most of their time in Run state. But when the server is running with one processor and saturated (400 clients) HttpProcessors spent 47% of their time in Ready state. This fact confirms our assertion that the server cannot handle all the incoming full SSL handshakes with only one processor.

It was predicted that when the server is saturated and running with two or four processors, the HttpProcessors would spend most of their time in Ready state (waiting for a free processor to execute), in the same way as when running with one processor. But, as Figure 3.23 makes clear, when the server was running with two processors and saturates, although the time spent on Ready state increased, the HttpProcessors spent 70% of their time in Blocked Synch state (blocked waiting for the synchronization with other HttpProcessors in a monitor). This can be due to the saturation of available processors on multiprocessor systems, as was the case here. When running with four processors, the time spent in Ready state and Blocked Synch state also increased.

Notice that, although the cause of the server saturation is the same when running with one, two or four processors (there are not enough processors to support demanded

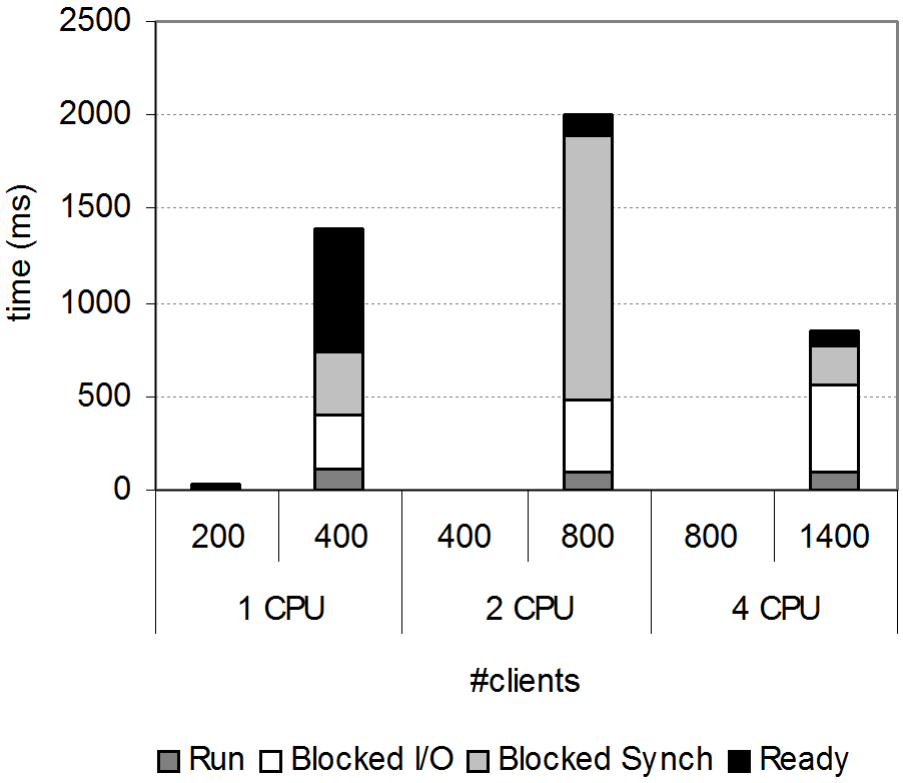


Figure 3.23 State of HttpProcessors when they are in the SSL handshake phase of a connection

computation), this saturation is manifested in different forms (waiting for a processor to become available in order to execute, or in a contention situation produced by the saturation of processors).

With the analysis performed, we can conclude that the processor is a bottleneck for Tomcat performance and scalability when running dynamic web applications in a secure environment. We have demonstrated that running with more processors allows the server to handle more clients before saturating, and even when the server reaches a saturated state, better throughput can be obtained by running with more processors.

3.3 Configuration Complexity

In the first section of this chapter we studied and evaluated multi-threaded and event-driven web server architectures. In the second section we characterized the performance and scalability of the multi-threaded web server under a secure workload. In both sections we evaluated the performance of the multi-threaded architecture with different benchmarks, including static web applications, dynamic web applications and secure web

applications. For each of the workloads evaluated we looked for the best multi-threaded web server configuration. **This is a slow and error-prone process that must be repeated each time the workload or the hardware configuration changes.**

From our experience and other related works [78] [35] we have identified the configuration parameters that most determine the performance of a generic multi-threaded web server. These parameters are the number of worker threads and the connection keep-alive timeout. The first parameter determines the concurrency level of the web server. A small number of worker threads usually limits the performance and scalability of the web server because only a limited number of clients can be served simultaneously. On the other hand, a large number of worker threads increase the maximum number of active clients but also increase the internal server contention for shared resources, as well as the web server memory usage. The second parameter, the keep-alive timeout, determines the maximum time that a worker thread will wait for a client request on a given active connection. If a request does not arrive in the specified connection keep-alive timeout the worker thread will close the connection to be able to serve other clients. A large value of the keep-alive timeout can reduce the effective number of available worker threads to serve client request because the worker threads can be blocked on inactive connections for long periods of time. On the other hand, small keep-alive timeouts allow for the fast switch of available threads to serve all active clients, but this small keep-alive timeout also produces an unnecessary overhead on the management of client connections, which can be very counterproductive for secure workloads that have an expensive connection handshake penalty.

In Section 3.1.5 we determined that a large keep-alive timeout (15 seconds of keep-alive timeout compared with the average 7 seconds of the client think time produced by Httperf) and a large number of threads is the best configuration for the Apache multi-threaded web server. The best performance was obtained with 6000 threads, but this large number of threads produced instabilities in the system that hung-up the machine several times. To avoid this undesirable situation we finally ran the test with the Apache web server configured with 4000 threads. In Section 3.2.6 we evaluated a secure dynamic workload with the multi-thread Tomcat web server. With this setup, we found that the best configuration is a 20 seconds keep-alive timeout and a moderate number of worker threads that dynamically vary form 200 to 300. We need to take into account that the Tomcat web server has an integrated heuristic that proportionally reduces the keep-alive timeout in function of the available worker threads. It is worth noting the huge difference (4000 vs 300) in the optimal number of threads for different workloads (static vs. dynamic). This is explained by the higher lock and resource contention in the dynamic web workload.

An inappropriate web server configuration can dramatically impact the performance of a multi-threaded web server. This problem is aggravated by the interaction between the web application type, the dynamic nature of the Internet and the error-prone process of manually configuring a web server. In complex environments like web server farms this problems is further worsen by factors such as heterogeneity of hardware, several web applications per web server and the burst nature of the Internet. In Section 4.3 we will study this problem and suggest how it can be mitigated by using the hybrid architecture presented in the next chapter.

3.4 Summary

In the first part of the chapter we presented the potential benefits of using an event-driven architecture on web servers design. Specifically, we studied how the new non-blocking I/O API (NIO), included in the J2SE platform since its 1.4 release, can help in the development of high-performance event-driven Java web servers. Our experimental NIO-based server outperformed a widely used commercial native-compiled web server, but reduced the complexity of the code and the system resources required to run it. The NIO API applied to the creation of event-driven Web Servers scales well when changing the workload intensity and when changing the number of processors and it performs at least as well as the Apache 2 server. Choosing a Java event-driven architecture to implement next-generations high-performance web servers is an option, but the really interesting conclusion of this study is that Java application servers no longer need to use native-compiled web servers as their web interfaces. Many commercial application servers choose Apache as their web interface because it is supposed to offer a better performance than a pure Java web server. But these decisions compromise the portability of the entire middleware. Choosing a Java event-driven architecture as the web interface for a Java application server can facilitate the integration of both elements, web server and application server, reducing the complexity of the whole system without paying the price of low performance. Despite the evident benefits in performance of the event-driven architecture, the integration of a non-blocking web server with a blocking application server remains a challenging task, which will be studied in chapter 4.

In the second part of the chapter we showned the impact of security on web server performance. First, we studied general multi-threaded web server behavior when using secure connection as a function of the number of clients. We determined that the main problem of multi-threaded web servers using SSL to provide security is the high number of full SSL connections established, because this kind of connection requires

a large amount of computation, producing server throughput degradation when a lot of connections are requested simultaneously. Second, we evaluated the effect of some client behaviors relative to SSL on server performance. These evaluation revealed that different client patterns (Do the clients reuse the SSL session IDs?, Do the clients retry the erroneous requests?, How long do the clients wait for a server response?) can have a decisive effect on server throughput. Following on from this, we presented a complete characterization of Tomcat application server scalability when executing the RUBiS benchmark using SSL, which is very valuable considering the few related work in this topic. This characterization was divided into two parts. First, we measured Tomcat vertical scalability (i.e. adding more processors) when using SSL and we analyzed the effect of this addition on server scalability. The results confirmed that running with more processors makes the server able to handle more clients before saturating and even when the server has reached a saturated state, better throughput can be obtained when running with more processors. Second, we analyzed the causes of server saturation when running with different numbers of processors using a performance analysis framework. This framework allows a fine-grain analysis of dynamic web applications by considering all levels involved in their execution. Our analysis revealed that the processor is a bottleneck for Tomcat performance on secure environments and that upgrading the system by adding more processors would improve the server scalability.

The results obtained in this work demonstrate the convenience of incorporating some kind of overload mechanism to the Tomcat multi-threaded web server, so as to avoid the throughput degradation produced by the arrival of a massive number of full SSL connections. A possible solution to the problem of secure connections is to make it possible for the web server to differentiate full SSL connections from resumed SSL connections (thus limiting the acceptance of full SSL connections to the maximum number acceptable without saturating), while accepting all the resumed SSL connections to maximize the number of client sessions successfully completed. As we will see in Section 4.2 the Hybrid architecture, which will be introduced in Chapter 4, naturally solves this problem without the need to add any ad-hoc mechanism to differentiate between full and resumed SSL connections. Whichever approach is used, we must recognize that security is an important issue which can seriously affect the scalability and performance of web applications.

Finally, in the last section of this chapter, we summarized the problems that we have encountered related to optimally configuring a multi-threaded web server. These issues will be studied in detail in Section 4.3 where the multi-threaded and our proposed Hybrid web server architectures are compared under three different workloads.

Chapter 4

The Hybrid Web Server Architecture

In Chapter 3 we studied the two most widely-used web server architectures: the multi-threaded and the event-driven architectures. We demostrated the superior performance and scalability of the event-driven architecture. Moreover, we identified a major degradation of performance on multi-threaded web servers that use secure connections. Finally, we introduced problems related to optimally configuring a multi-threaded web server for optimal performance. Despite the problems that we have identified, the multi-threaded web server architecture is the most widely-used type of current web servers. In contrast, the event-driven architecture, which has better performance and scalability properties, is normally used only as a high performance front end of more complex multi-threaded web servers. This configuration, a high performance event-driven web server that serves the static content and forward dynamic request to a multi-threaded web server, is quite usual, but it is not optimal from the point of view of performance, or from the point of view of configuration complexity or manageability. To deal with all these issues, we present the Hybrid architecture, which -as its name implies- mixes concepts of both the multi-threaded and the event-driven architectures. In this chapter we present a detailed description of this architecture, comparing its performance with the multi-threaded architecture. It will be seen that the hybrid architecture outperforms the multi-thread architecture in quantitative and, more important, qualitative terms. Finally, we show how the hybrid architecture dramatically reduces the configuration complexity of a web server, converting the web server configuration into a trivial task.

4.1 Introduction to the Hybrid Architecture

4.1.1 Motivation

The performance of an e-commerce application can be measured according to technical metrics but also following business indicators. The revenue obtained by a commercial web application is directly related to the amount of clients that complete business transactions. In technical terms, a business transaction is completed when a web client successfully finishes a browsing session. In this chapter we introduce a novel web server architecture that combines the best aspects of both multi-threaded and event-driven architectures, the two major existing alternatives, to create a server model that offers an improved performance in terms of user session completions without loosing the natural ease of programming characteristic of the multi-threading paradigm. We describe the implementation of this architecture on the Tomcat 5.5 server and evaluate its performance. The results obtained demonstrate the feasibility of the hybrid architecture

and the performance benefits that this model affords for e-commerce applications

4.1.2 Introduction

Most e-commerce applications are distributed applications based on the well-known client/server paradigm that reside mostly in an application server and that are usually accessed by a remote thin web client. The communication protocol between the server and the client is the Hypertext Transfer Protocol (HTTP), which in the server side is parsed and processed by a component of the application server that is commonly known as the web container. In most cases, a web container can be considered as a web server that can support some language extensions to create more flexible web applications.

The performance of an e-commerce application can be measured in technical terms or following business indicators (see [58] for further discussion of this topic). Usually, the technical measurement units for a web server are replies per second (throughput) and the response time. In contrast, most performance metrics based on business indicators can be reduced to one: profit. The revenues a website can generate are directly related to the amount of commercial transactions the website can complete. In general, a commercial transaction is completed when a user browsing session successfully finishes. Therefore, the web container architectures must be designed to obtain a high performance in user sessions units in order to satisfy the needs of e-commerce applications.

The architectural design of most currently existing web containers follow the multi-threading paradigm (Apache and Tomcat [39] are widely-used examples) because it leads to a natural ease of programming and simplifies the development of the web container logic. Unfortunately, this model does not obtain high performance in terms of user session completions. Alternatively, an event-driven model (used in Flash[67] and in the SEDA[85] architecture) can be adopted to develop a web server container. This model solves some of the problems present in the multi-threaded architecture but transforms the development of the web container into a more difficult task.

In this chapter we introduce a new hybrid web server architecture that exploits the best of each of the two discussed server architectures. With this hybrid architecture, an event-driven model is applied to receive the incoming client requests. When a request is received, it is serviced following a multi-threaded programming model, with the resulting simplification of the web container development associated with the multi-threading paradigm. When the request processing is completed, the event-driven model is applied again to wait for the client to issue new requests. In this way, the best features of each model are combined and, as is discusses in the following subsections, the performance of the server is remarkably improved in terms of user session completions.

The rest of the chapter is structured as follows: Subsection 4.1.3 discusses the characteristics of the two architectures involved in the creation of a hybrid web server, Subsection 4.1.4 describes the implementation details of the hybrid web server, and in Subsection 4.1.5, we present the execution environment where the experimental results presented in this work were obtained. Finally, Subsection 4.1.6 presents the experimental results obtained in the evaluation of the hybrid web server and subsection 4.1.7 makes some concluding remarks and discusses some of the future work suggested by this study.

4.1.3 Web server architectures

There are multiple architectural options for a web server design, depending on the concurrency programming model chosen for the implementation. The two major alternatives are the multi-threaded model and the event-driven model. In both models, the work tasks to be performed by the server are divided into work assignments that are assumed each one by a thread (a worker thread). If a multi-threaded model is chosen, the unit of work that can be assigned to a worker thread is a client connection, which is achieved by creating a virtual association between the thread and the client connection which is not broken until the connection is closed. Alternatively, in an event driven model the work assignment unit is a client request, so there is no real association between a server thread and a client.

The multi-threaded programming model leads to a very easy and natural way of programming a web server. The association of each thread with a client connection results in a comprehensive thread life-cycle, which starts with the arrival of a client connection request and finishes with the connection close. This model is especially appropriate for short-lived client connections and with low inactivity periods, which is the scenario created by the use of non persistent HTTP/1.0 connections. A pure multi-threaded web server architecture is generally composed of an acceptor thread and a pool of worker threads. The acceptor thread is in charge of accepting new incoming connections, after which each established connection is assigned to one thread of the workers pool, which will be responsible for processing all the requests issued by the corresponding web client.

The introduction of connection persistence in the HTTP protocol, already in the 1.0 version of the protocol but mainly with the arrival of HTTP/1.1, had a big impact on the performance of existing multi-threaded web servers. Persistent connections, which means connections that are kept alive by the client between two successive HTTP requests that in turn can be separated in time by several seconds of inactivity (think times), mean that many server threads can be retained by clients even when no requests are being issued and the thread remains in idle state. The use of blocking I/O operations on the sockets

is the cause of this performance degradation scenario. The situation can be solved by increasing the number of threads available (which in turn results in a contention increase in the shared resources of the server that require exclusive access) or by introducing an inactivity timeout for the established connections, which can be reduced as the server load is increased. When a server is put under a severe load, the effect of applying a shortened inactivity timeout to the clients is a virtual conversion of the HTTP/1.1 protocol into the older HTTP/1.0, with the consequent loss of the performance of the connection persistence.

In this model, the effect of closing client connections to free worker threads reduces the probability of a client completing a session to nearly zero. It is especially important when the server is under overload conditions, where the inactivity timeout is dynamically decreased to the minimum possible in order to free worker threads as quickly as possible, which in turn results in all the established connections being closed during think times. This causes a higher competition among clients trying to establish a connection with the server. If we extend this to the length of a user session, we find that the probability of finishing a session successfully with this architecture is still much lower than the probability of establishing each one of the connections it is composed of. Consequently, the server achieves a very low performance, in terms of session completions. This situation can be alleviated by increasing the number of worker threads available in the server, but this measure also produces a significant increase in the internal web container contention, with a corresponding performance slowdown.

The event-driven architecture completely eliminates the use of blocking I/O operations for the worker threads, reducing their idle times to the minimum because no I/O operations are performed for a socket if no data is available to be read. With this model, maintaining a large number of clients connected to the server does not represent a problem because one thread will never be blocked waiting for a client request. In this way, the model detaches threads from client connections, and only associates threads to client requests, considering them as independent work units. An example of web server based on this model is described in [67], and a general evaluation of the architecture can be found in [20].

In an event driven architecture, one thread is in charge of accepting new incoming connections. When the connection is accepted, the corresponding socket channel is registered in a channel selector where another thread (the request dispatcher) will wait for socket activity. Worker threads are only awakened when a client request is already available in the socket. When the request is completely processed and the reply has been successfully issued, the worker thread registers again the socket channel in the selector

and is free to be assigned to new client requests. This operation model avoids worker threads being blocked in socket read operations during client think times and eliminates the need for connection inactivity timeouts and their associated problems.

A remarkable characteristic of the event-driven architectures is that the number of active clients connected to the server is unbounded, so an admission control[31] policy must be implemented. Additionally, as the number of worker threads can be very low (one should be enough) the contention inside the web container can be reduced to the minimum.

Hybrid architecture

In this chapter we propose a hybrid architecture that takes advantage of the strong points of both discussed architectures, the multi-threaded and the event driven. In this hybrid architecture, the operation model of the event-driven architecture is used for the assignment of client requests to worker threads (instead of client connections) and the multi-threaded model is used for the processing and service of client requests, where the worker threads will perform blocking I/O operations when required. This architecture can be used to decouple the management of active connections from the request processing and servicing activity of the worker threads. In this way, the web container logic can be implemented following the multi-threaded natural programming model and the management of connections can be done with the highest possible performance, without blocking I/O operations and achieving a maximum overlapping of the client think times with the processing of requests.

In this architecture the acceptor thread role and the request dispatcher role from the pure event-driven model are used while, the worker thread pool, which perfors blocking I/O operations when necessary, from the pure multi-threaded design is also used. This makes it possible for the hybrid model to avoid the need to close connections to free worker threads without renouncing to the multi-threading paradigm. In consequence, the hybrid architecture makes better use of the characteristics introduced to the HTTP protocol in the 1.1 version, such as connection persistence, with the corresponding reduction in the number of client reconnections (and the corresponding bandwidth save). Additionally, and as has been explained in our discussion of the event-driven architectures, some kind of admission control policy must be implemented in the server in order to maintain the system an acceptable load level.

4.1.4 Hybrid architecture implementation

To validate the proposed hybrid architecture we implemented it inside Tomcat 5.5, a widely-used web container. Tomcat 5 is built on the top of the Java platform, which provides non blocking I/O facilities across different operating systems in its NIO (Non Blocking I/O) API. The implementation was done by modifying a pluggable component of the server generally named connector which is the server component in charge of handling communications with the client

Tomcat is an open-source servlet container developed under the Apache license. Its primary goal is to serve as a reference implementation of the Sun Servlet and JSP specifications, and to be a quality production servlet container. In this chapter, two major components of Tomcat are considered: Coyote and Catalina. Coyote is the default Tomcat connector and deals with client connection request parsing and thread pooling. Catalina is a servlet container, and implements most of the web container logic. The implementation of the hybrid server architecture in Tomcat only affects Coyote.

Coyote follows a connection service schema where, at a given time, one thread (an HttpProcessor) is responsible for accepting a new incoming connection on the server listening port and assigning to it a socket structure. From this point on, this HttpProcessor will be responsible for attending and parsing the received requests through the persistent connection established with the client, while another HttpProcessor will continue to accept new connections. HttpProcessors are commonly chosen from a pool of threads in order to avoid thread creation overheads. A connection timeout is programmed to close the connection if no more requests are received in a period of time. When a request is parsed, Coyote requires Catalina to process the request and to send the corresponding response to the client.

The implementation of the hybrid architecture changes the original Coyote threading and I/O model. One thread is in charge of accepting and registering, through a NIO selector, new incoming connections. When a registered connection becomes active (i.e. it has data available to read so a read operation over the socket will not be blocking it), it is dispatched by the selector thread to a small pool of HttpProcessor threads. Each HttpProcessor services only one request for each assigned active connection. The request is read and parsed, always without blocking the thread, and it is sent to Catalina which processes the request. When the request is finished, the connection is re-registered on the selector and the thread is sent back to the pool until a new active connection is assigned to it.

A major drawback of this implementati is that when the system becomes overloaded the acceptor thread allows new connections to enter the server faster than Catalina can

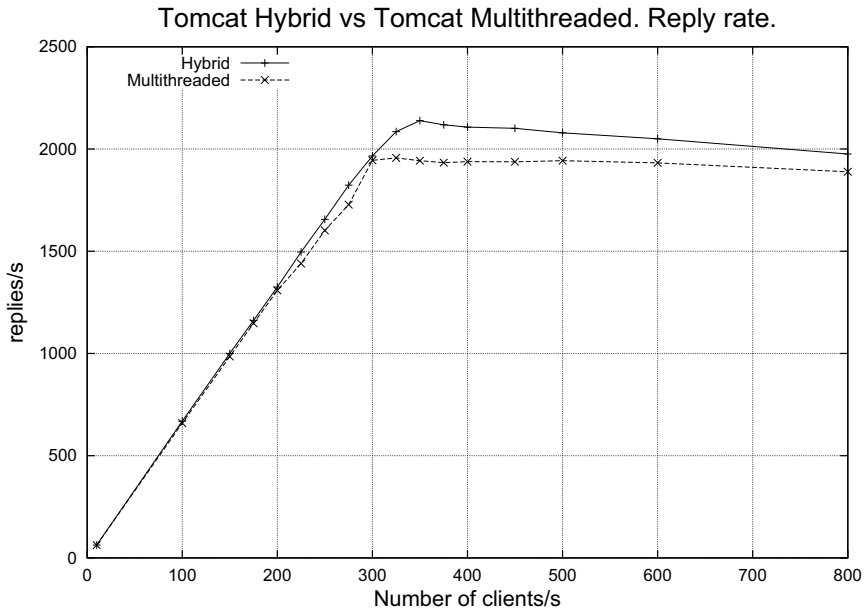


Figure 4.1 Throughput comparison under an static content workload

service them, which results in a rapid growth of the number of concurrent connections, which in turn causes a severe response time degradation. As a result, the number of client timeouts grows an the throughput decreases. To avoid this problem we have introduced a simple but effective admission control mechanism (similar to the back-pressure technique described in [85]), that prevents the acceptor thread from accepting new connections while all HttpProcessors are busy.

4.1.5 Testing environment

The hardware platform for the experiments presented in this chapter is composed of a 4-way Intel Xeon 1.4 GHz with 2GB RAM to run the web servers and a 2-way Intel XEON 2.4 GHz with 2 GB RAM to run the benchmark clients. For the benchmark applications that require the use of a database server, a 2-way Intel XEON 2.4 GHz with 2 GB RAM was used to run MySQL v4.0.18, with the MM.MySQL v3.0.8 JDBC driver. All the machines were running a Linux 2.6 kernel, and were connected through a switched Gbit network. The SDK 1.5 from Sun was used to develop and run the web servers.

The servers were tested in two different scenarios, one to evaluate the server performance for a static content application and another for a dynamic content environment. The workload for the experiments was generated using a workload generator and web performance measurement tool called Httperf[62]. This tool permits the creation of a continuous flow of HTTP requests issued from one or more client machines and processed

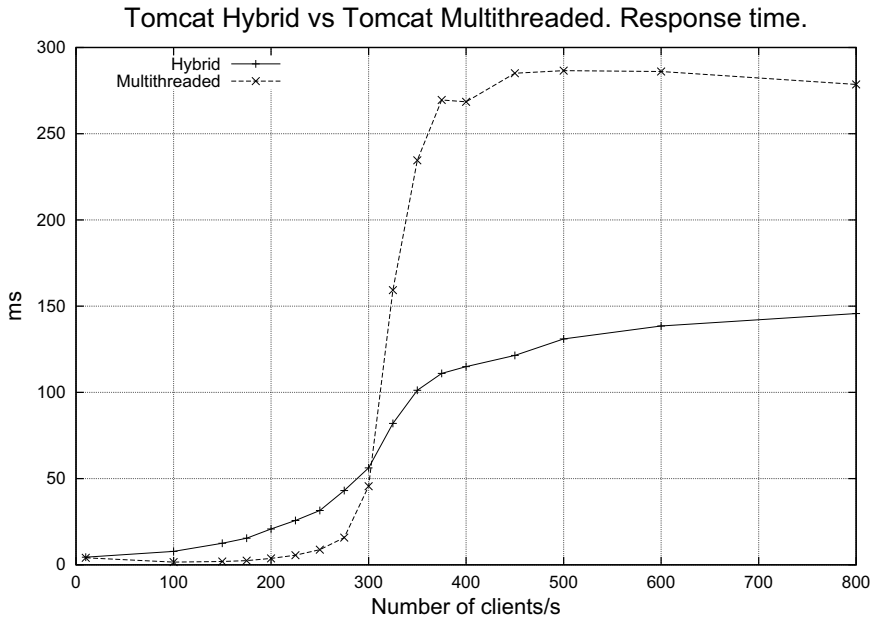


Figure 4.2 Response time under an static content workload

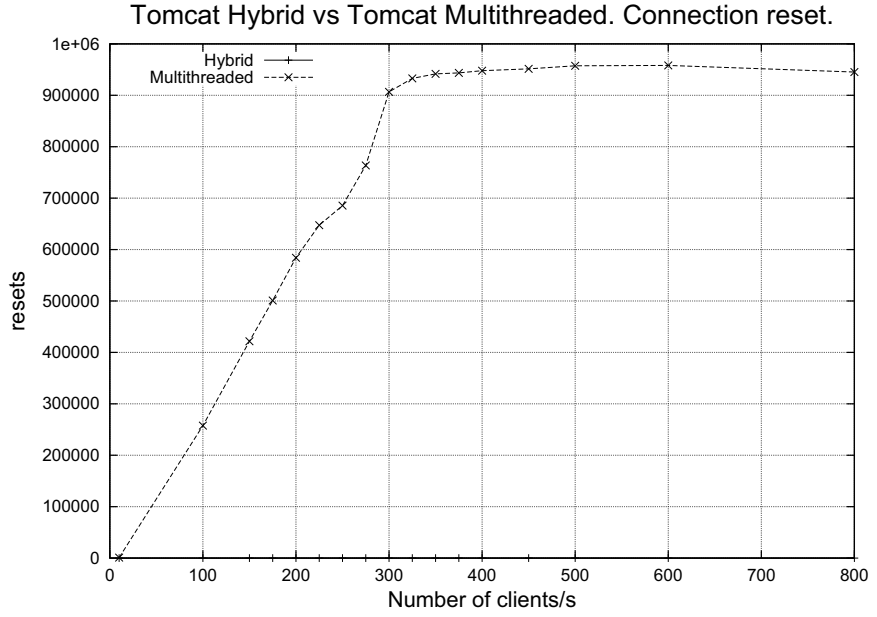


Figure 4.3 Number of connections closed by the server by a timeout expiration

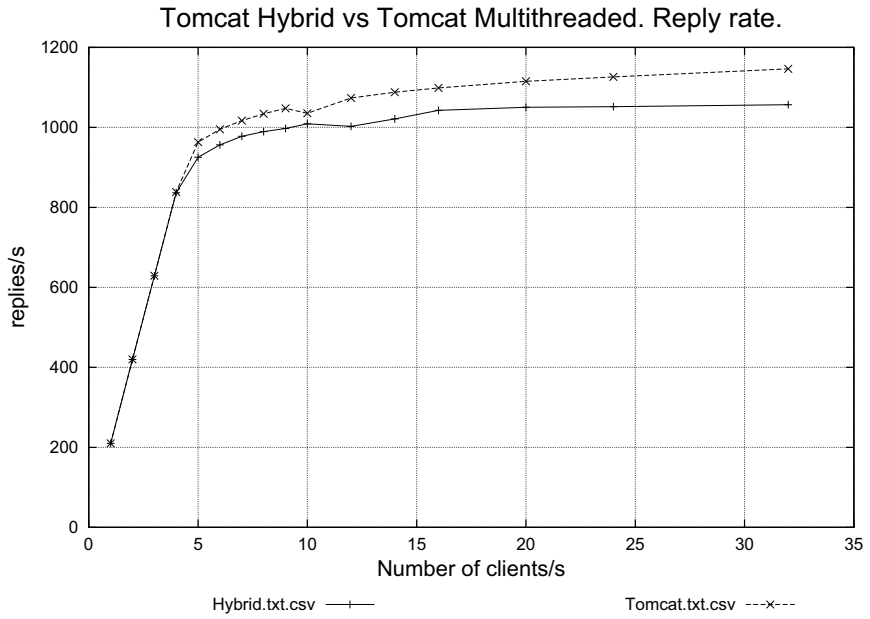


Figure 4.4 Reply throughput comparison under a dynamic content workload

by one server machine: the SUT (System Under Test). The configuration parameters of the benchmarking tool used for the experiments presented in this chapter were set to create a realistic workload, with non-uniform reply sizes, and to sustain a continuous load on the server. One of the parameters of the tool represents the number of clients/s, i.e.the load. Each emulated client opens a session with the server. The session remains alive for a period of time, called session time, at the end of which the connection is closed. Each session is a persistent HTTP connection with the server, used by the client to repeatedly send requests, some of them pipelined. The requests issued by Httperf were extracted from the surge[17] workload generator for the static content scenario and from the RUBiS [15] application for the dynamic content environment.

A static content application is characterized by the short length of the user sessions as well as by the low computational cost of each request to be serviced. This is the scenario reproduced with the Surge[17] workload generator for these experiments. The request distribution produced by Surge is based on the observation of some real web server logs, from where a data distribution model of the observed workload was extracted. This methodology guaranties than the used workload for the experiments follows a realistic load distribution.

A dynamic content application is characterized by the long length of the user sessions (an average of 300 requests per session in contrast to the 6 requests per session for the static workload) as well as by the high computational cost of each request to be

serviced (including embedded requests to external servers, such as databases). For our experiments, the chosen dynamic content benchmark application was RUBiS[15] (Rice University Bidding System), in its 1.4 servlets version. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. The workload distribution generated by Httperf was extracted from the RUBiS client emulator, which uses a Markov model to determine which subsequent link from the response to follow. Each emulated client waits for an amount of time, before initiating the next interaction, emulating in this way the "thinking times observed in real clients. Httperf also permits configuring a client timeout. If this timeout elapses and no reply has been received from the server, the current session is discarded.

4.1.6 Experimental results

In our experiments we evaluated the performance characteristics of each web server architecture, the original multi-threaded Tomcat and our hybrid Tomcat implementation, when used in the two different scenarios already explained in Subsection 4.1.5: a static content application (Surge) and a dynamic content one (RUBiS). For the experiments, we configured Httperf setting the client timeout value to 10 seconds, and used the configuration described in Subsection 4.1.5. Each individual benchmark execution had a fixed duration of 30 minutes for the dynamic content tests and 10 minutes for the static content experiments.

Static content

The first performance metric evaluated for this scenario was the throughput obtained for each architectural design, measured in replies per second. The results for both architectures are shown in Figure 4.1, where it can be seen that the multi-threaded architecture obtains a slightly lower performance than the hybrid one in terms of throughput. The results are so close, however, that they can be considered equivalent. It is remarkable that the same throughput is obtained by the hybrid architecture with a thread pool size of only 10 threads, while the multi-threaded architecture requires 500 threads to obtain the same results.

Moving from, the throughput to the response time observed for each implementation, Figure 4.2 clearly shows that the hybrid architecture offers much better results than the multi-threaded one. When the system is not saturated (under a load equivalent to 300 new clients per second), the response time for the multi-threaded server is slightly better than for the hybrid design, possibly because of the overhead introduced by the extra operations

that the hybrid server must do to register the sockets in the selector and to switch them between blocking and non-blocking mode when moving from multi-threaded to the event-driven and reverse. This effect would be dispelled in a WAN environment, where the latencies are much higher than in the Gbit LAN used for the experiments. When the system is saturated, over 300 new clients/s, the benefits of the hybrid architecture becom apparent and the response time reduction with respect to the multi-threaded version of the server is about 50%, falling from a 300 ms average response time for the multi-threaded architecture to a 150 ms response time for the hybrid design.

Another interesting difference between the behavior of the two studied architectures can be observed in Figure 4.3, which shows the number of connections that were closed by the server because the client inactivity period was too long. This caused, the server timeout to expire and obligated the client to be reconnected to resume its session. As can be seen, while the hybrid architecture did not produce this kind of situation (zero errors are detected by the benchmark client), a high number of errors were detected for the multi-threaded architecture. The explanation for this difference is that the multi-threaded design needs free worker threads to make them available for new incoming connections. This means that client inactivity periods must be avoided by closing the connection and requiring the client to resume its session with a new connection establishment when necessary. In the hybrid architecture, which assigns client requests as work units to the worker threads instead of client connections, this situation is naturally avoided and the cost of keeping a client connection event in periods of inactivity is equivalent to the cost of keeping the connection socket opened. The effect of this characteristic of the hybrid architecture is that all reconnections are eliminated.

Dynamic content

Dynamic content applications implement a more complex and more developed semantics than static ones, which usually implies longer user sessions and means that common performance metrics are partially redefined in terms of business concepts. E-commerce applications are naturally more concerned with sales or business transactions than with more technical metrics such as the throughput or the response time offered by the server.

For this experiment, we assumed that one of the most important metrics for an auction website (the scenario reproduced by RUBiS, see subsection 4.1.5 for more details) is the number of user sessions that are completed successfully. Each user that can complete its navigation session represents a potential bid for an item and in consequence a higher profit for the auction company.

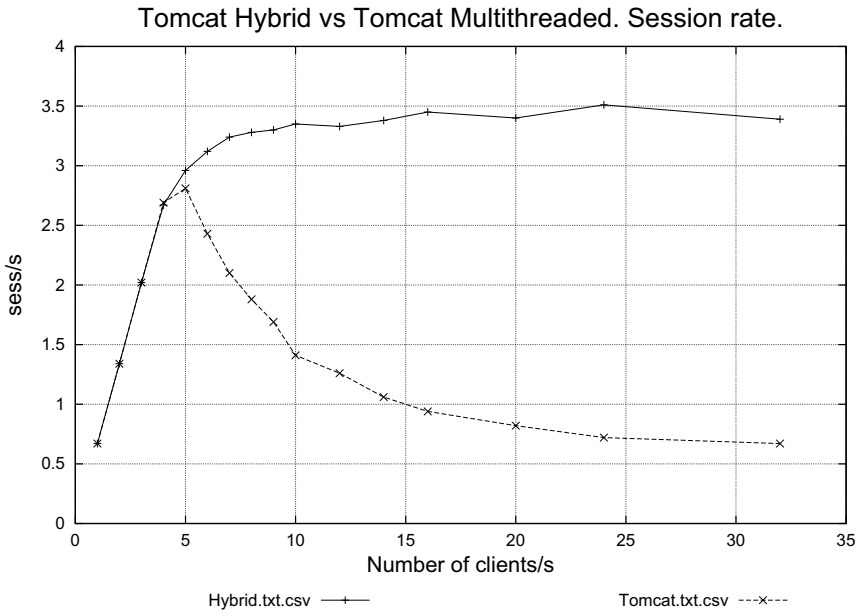


Figure 4.5 Successfully completed session rate under a dynamic content workload

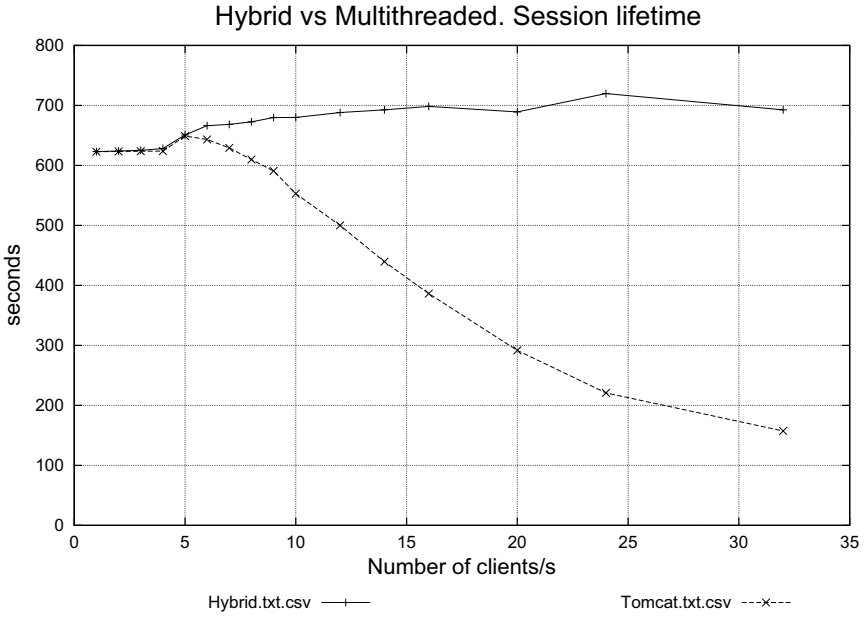


Figure 4.6 Lifetime comparison for the sessions completed successfully under a dynamic content workload

In Figure 4.4, it can be seen that the throughput offered by the two architectural designs is very similar, although the multi-threaded architecture shows a slightly better performance when the server is saturated.

Looking at the number of sessions completed per second, as presented in Figure 4.5, it can be seen that the number of successfully finished sessions reached by the hybrid architecture is substantially higher than the number reached by the multi-threaded design, especially beyond saturation. Figure 4.5 shows that, in the multi-threaded architecture the number of completed sessions per second tends to fall as the workload intensity increases. This is because, under a pure multi-threaded design, the worker threads are obliged to close the client connections in order to be freed and remain available for new incoming connection requests. This situation, under high loads, leads to a scenario where the clients whose connections have been closed by the server, find it increasingly difficult to be reconnected because the amount of active clients trying to establish a connection is remarkably higher than the amount of worker threads available in the server. If this is extended to the amount of connections required to complete a long user session, characteristic of dynamic applications, the probability of being able to finish the session successfully is reduced to near zero, as has already been discussed in subsection 4.1.3.

This explanation to the difference of performance between the two architectural designs in terms of session completion is supported by the results shown in figure 4.6 that indicate that the sessions completed by the multi-threaded server are significantly shorter than the sessions completed by the hybrid server. This explains that the reply rate for the multi-threaded server can be sustained even when the session rate is remarkably reduced because it proves that the sessions completed by the multi-threaded server are those ones with less requests (the shorter ones). Completing only short sessions means that many active clients finishes their sessions unsuccessfully, which in turn may result in an important amount of unsatisfied clients that have received a very poor quality of service.

4.1.7 Summary

In this section we have shown how a hybrid web server architecture that combines the best of a multi-threaded design with the best of an event-driven model can be used to obtain a high performance web server, especially when the throughput is measured in successfully completed user sessions per second. The proposed implementation in the Tomcat 5.5 code offers a slightly better performance than the original multi-threaded Tomcat server when it is tested for a static content application, and a greatly enhanced performance when used with dynamic content application, where each user session failure

is related to business revenue losses. Additionally, the hybrid version allows the natural way of programming introduced by the multi-threading paradigm to be maintained for most of the web container code.

Further research will be done into ways of exploiting the advantages of the hybrid architecture benefits in the area of session based admission control, especially in the presence of secure connections (SSL), where the cost of client reconnections have an enormous impact on performance (see Section 3.2 for more details on this). In the next section we will study the performance and behavior of the Hybrid architecture with secure dynamic applications. The hybrid architecture also reduces web server complexity, as we will see in section 4.3, which is an important step toward the implementation of autonomic servers[54].

4.2 The Hybrid Architecture under Secure Workloads

Nowadays the success of many e-commerce applications, such as on-line banking, depends on their reliability, robustness and security. Designing a web server architecture that can maintain these properties under high loads is a challenging task because they are naturall constraints to performance. As we have seen in Section 3.2, the industry standard for providing security on web applications is the use the Secure Socket Layer (SSL) protocol which creates a secure communication channel between the clients and the server. Traditionally, the use of data encryption has resulted in a negative performance impact on web application servers because it is an extremely CPU consuming task, reducing the throughput achieved by the server as well as increasing its average response time. Given that the revenue obtained by a commercial web application is directly related to the amount of clients that complete business transactions, the performance of such secure applications becomes a critical objective for most companies. In this chapter we evaluate a novel hybrid web server architecture (implemented over Tomcat 5.5) that combines the best aspects of the two most widely-used server architectures, the multi-threaded and the event-driven, to provide an excellent trade-off between reliability, robustness, security and performance. The results obtained demonstrate the feasibility of the proposed hybrid architecture as well as the performance benefits that this model introduces for secure web applications, providing the same security level as the original Tomcat 5.5 and improving reliability, robustness and performance, in terms of both technical and business metrics.

4.2.1 Introduction

Many e-commerce applications must offer a secure communication channel to their clients in order to achieve the level of security and privacy required to carry out most commercial transactions. But the cost of introducing security mechanisms in on-line transactions is not negligible. The industry standard for secure web communications is the Secure Socket Layer (SSL) protocol, which is generally used as a complement to the Hypertext Transport Protocol (HTTP) to create the Secure HTTP protocol (HTTPS). The primary goal of the SSL protocol is to provide privacy and reliability between two applications in communication over the Internet. This is achieved by using a combination of public and private cryptography to encode the communication between the peers.

The use of public key cryptography introduces an important computational cost to the web containers. Each time a new connection attempt is accepted by a server, a cryptographic negotiation takes place. This initial handshake is required by the peers in order to exchange the encryption keys that will be used during the communication. The cost of the initial handshake is so high that it severely limits the maximum number of new connections than can be accepted by the server in a period of time, as well as degrading the performance of the server to unacceptable levels, as can be seen in [41].

The architectural design of most existing web servers is based on the multi-threaded paradigm (Apache[81] and Tomcat [39] are widely-used examples), which assigns one thread to each client connected to the server. These threads, commonly known as worker threads, are in charge of attending all the requests issued by their corresponding client until it gets disconnected. The problem with this model is that the maximum number of concurrent clients accepted by the server is limited to the number of threads created. Confronted by this this situation, the solution adopted by most web servers is to impose an inactivity timeout on each connection client, forcing the connection to close if the client does not produce any work activity before the timeout expires.

The effect of closing client connections is relatively insignificant when working with plain connections, but it becomes extremely negative when dealing with secure workloads. Closing connections, especially when the server is overloaded, increases the number of cryptographic handshakes to be performed and drastically reduces the capacity of the server, which has an important negative impact on the maximum throughput achieved by the server. In contrast to this, an alternative architectural design for web servers is the event-driven model, already used in Flash[67] and in the SEDA[85] architecture. This model seeks to solve the problems associated to the multi-threading paradigm, especially in client-server environments. But this model lacks of the innate ease of programming associated to the multi-threading model, making the task of developing web servers far

more complex.

In this chapter we evaluate the performance of the hybrid web server architecture presented in Section 4.1 when used with a secure communication protocol which exploits the best of each of the two server architectures already discussed. To summarize, with this hybrid architecture, an event-driven model is used to receive the incoming client requests. When a request is received, it is serviced following a multi-threaded programming model, with the resulting simplification of the web container development associated to the multi-threading paradigm. When the request processing is completed, the event-driven model is applied again to wait for the client to issue new requests. In this way, the best feature of each model are combined and, as is discussed in following sections, the performance of the server improves enormously, especially in the presence of secure communication protocols.

The rest of the chapter is structured as follows: Section 4.2.2 describes the HTTP/s protocol, Section 4.1.3 discusses the characteristics of the multi-threaded, event-driven and hybrid architectures; in Section 4.2.3, we present the execution environment where the experimental results presented in this work were obtained and, finally, Section 4.2.4 presents the experimental results obtained in the evaluation of the hybrid web server and Section 4.2.5 gives some concluding remarks and discusses some of the future studies that may follow.

4.2.2 HTTP/S and SSL

As we have described in Section 3.2.3 the HTTP/S (HTTP over SSL) is a secure Web protocol developed by Netscape. HTTPS is really just the use of Secure Socket Layer (SSL) as a sub-layer under its regular HTTP application layering. The SSL protocol provides communications privacy over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. To achieve these objectives it uses a combination of public-key and private-key cryptography algorithm and digital certificates (X.509).

The SSL protocol does not introduce any new degree of complexity in web applications structure because it works almost transparently on top of the socket layer. However, SSL dramatically increases the computation time necessary to serve a connection, due to the use of cryptography to achieve its objectives. This increment has a noticeable impact on server performance, which has been evaluated in [41]. This study concludes that the maximum throughput obtained when using SSL connections is 7 times lower than when using normal connections. The study also makes clear that when the server is attending non-secure connections and saturates, it can maintain the throughput if new clients arrive,

while if it is attending SSL connections, the saturation of the server provokes a serious degradation of the throughput.

The SSL protocol fundamentally has two phases of operation: SSL handshake and SSL record protocol. We offer an overview of the SSL handshake phase, which is responsible for most of the computation time required when using SSL. A detailed description of the whole protocol can be found in RFC 2246 [36]. The SSL handshake allows the server to authenticate itself to the client using public-key techniques such as RSA, and then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server. Two different SSL handshake types can be distinguished: the full SSL handshake and the resumed SSL handshake. The full SSL handshake is negotiated when a client establishes a new SSL connection with the server, and requires the complete negotiation of the SSL handshake. This negotiation includes parts that spend a lot of computation time to be accomplished. We have measured the computational demand of a full SSL handshake in a 1.4 GHz Xeon machine to be around 175 ms.

The SSL resumed handshake is negotiated when a client establishes a new HTTP connection with the server but using an existing SSL connection. As the SSL session ID is reused, part of the SSL handshake negotiation can be avoided, which considerably reduces the computation time for performing a resumed SSL handshake. We have measured the computational demand of a resumed SSL handshake in a 1.4 GHz Xeon machine to be around 2 ms. Notice the big difference between negotiating a full SSL handshake and negotiating a resumed SSL handshake (175 ms versus 2 ms).

Based on these two handshake types, two types of SSL connections can be distinguished: the new SSL connections and the resumed SSL connections. The new SSL connections try to establish a new SSL session and must negotiate a full SSL handshake. The resumed SSL connections can negotiate a resumed SSL handshake because they provide a reusable SSL session ID (they resume an existing SSL session).

4.2.3 Testing environment

The hardware platform for the experiments presented in this chapter is composed of a 4-way Intel Xeon 1.4 GHz with 2GB RAM to run the web servers and a 2-way Intel XEON 2.4 GHz with 2 GB RAM to run the benchmark clients. For the benchmark applications that require the use of a database server, a 2-way Intel XEON 2.4 GHz with 2 GB RAM was used to run MySQL v4.0.18, with the MM.MySQL v3.0.8 JDBC driver. All the machines were running a Linux 2.6 kernel, and were connected through a switched Gbit

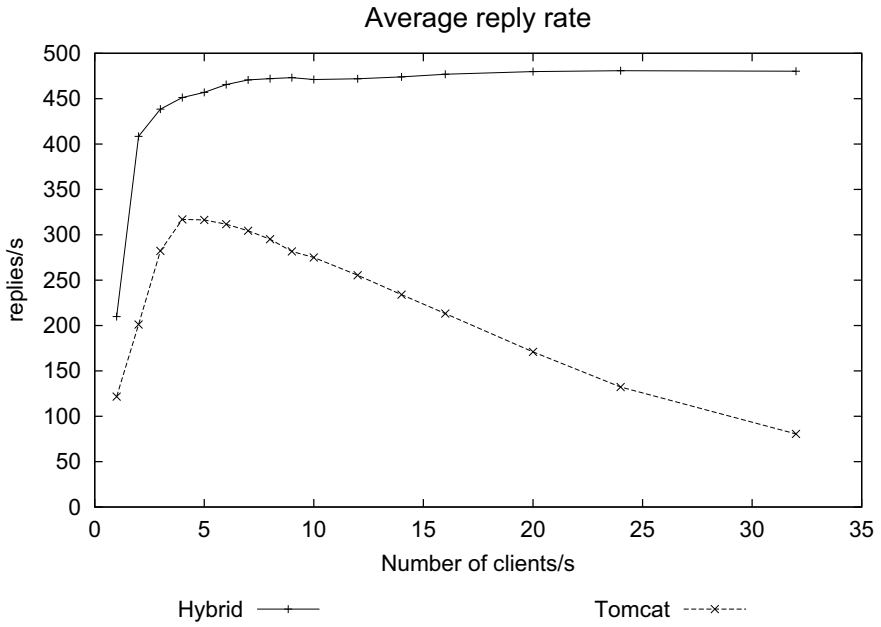


Figure 4.7 Reply throughput comparison between Tomcat server and hybrid server

network. The SDK 1.5 from Sun was used to develop and run the web servers. All the tests are performed with the common RSA-3DES-SHA cipher suit. Handshake is performed with 1024 bit RSA key. Record protocol uses triple DES to encrypt all application data. Finally, SHA digest algorithm is used to provide the Message Authentication Code (MAC).

The workload for the experiments was generated using a workload generator and web performance measurement tool called Httperf [62]. This tool, which supports both HTTP and HTTPS protocols, allows the creation of a continuous flow of HTTP/S requests issued from one or more client machines and processed by one server machine: the SUT (System Under Test). The configuration parameters of the benchmarking tool used for the experiments presented in this chapter were set to create a realistic workload, with non-uniform reply sizes, and to sustain a continuous load on the server. One of the parameters of the tool represents the number of new clients per second initiating an interaction with the server. Each emulated client opens a session with the server. The session remains alive for a period of time, called session time, at the end of which the connection is closed. Each session is a persistent HTTP/S connection with the server, used by the client to repeatedly send requests, some of them pipelined. The requests issued by Httperf were extracted from the RUBiS [15] application. A secure dynamic content application is characterized by the long length of the user sessions as well as by the high computational cost of the first connection (initial SSL handshake) and subsequent request to be serviced (including

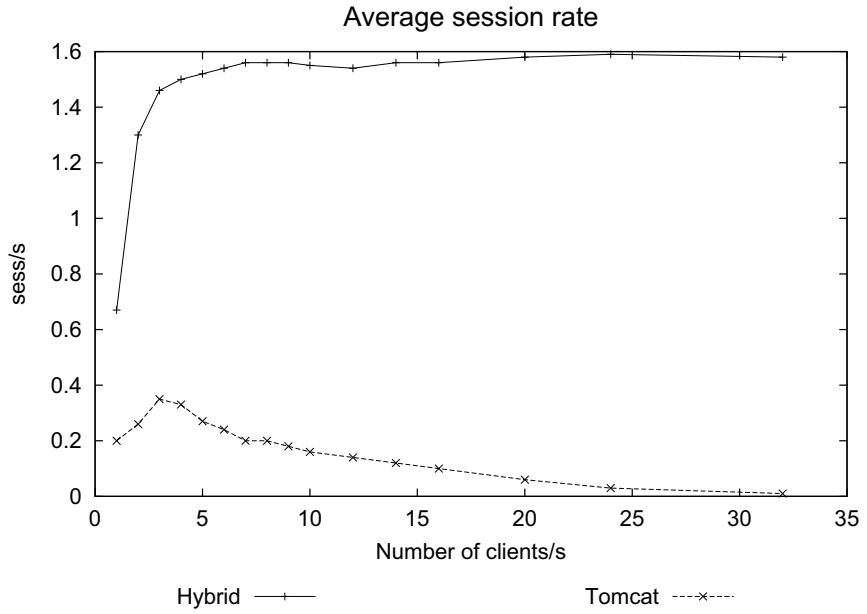


Figure 4.8 Session throughput comparison between Tomcat server and hybrid server

embedded requests to external servers, such as databases).

4.2.4 Experimental results

In this section we compare the performance of the proposed hybrid server architecture with the out-of-the-box Tomcat architecture, under a secure workload as well as under a plain workload.

The workload generator used for the experiments, Httperf, was configured using a client timeout value of 10 seconds and according to the configuration described in section 4.2.3. Each individual benchmark execution had a fixed duration of 30 minutes.

Secure dynamic content

From Figure 4.7, it can be seen that the throughput of the hybrid server is always better than the original Tomcat, moreover the difference between them increases as the load is increased. When the load is low and the original Tomcat architecture is below saturation, the benefits of the hybrid architecture are already evident. The use of the non-blocking I/O API (NIO), which offers a higher performance than the standard stream-based Java I/O, leads the hybrid architecture to offer a higher performance than the original Tomcat server. By the time the saturation point of the Tomcat server is reached, the hybrid architecture shows a throughput that is a 50% higher than the original multi-threaded

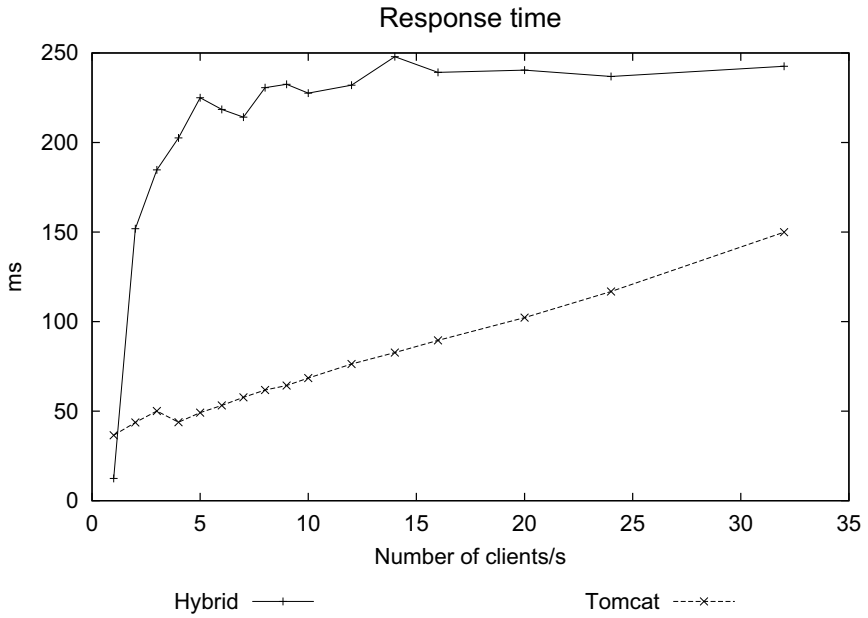


Figure 4.9 Response time comparison between Tomcat server and hybrid server

architecture. When the load is increased beyond the saturation point, the performance of the hybrid server remains almost constant while the out-of-the-box Tomcat server starts reducing its output level linearly with the load increase. The benefits of the hybrid architecture beyond the saturation point are explained by the higher use that the hybrid architecture makes of the connection persistence characteristics of the HTTP/1.1 protocol compared to the original multi-threaded approach. As more TCP connections are reused, more clients can keep their connections established and consequently less connection re-establishments will be required. When the considered workload uses data encryption, reducing the number of connection establishments is synonymous with less SSL handshakes negotiations and in consequence an important reduction of the processing requirements for the server system.

Usually, the workload produced over secure e-business applications is session-based. At the beginning of the session, the client gets connected. After that, the session requests are issued and correspondingly processed by the server. Finally, when the session is finished, the client gets disconnected and the user session is considered completed. The need of client re-connections introduced by the limitations of the multi-threaded server model makes it increasingly difficult to complete user sessions successfully. This effect is shown in Figure 4.10. As can be seen, when the saturation point is reached, the average length of the user sessions successfully completed for the original Tomcat server starts decreasing. The cause of this phenomenon is the higher number of re-connections needed

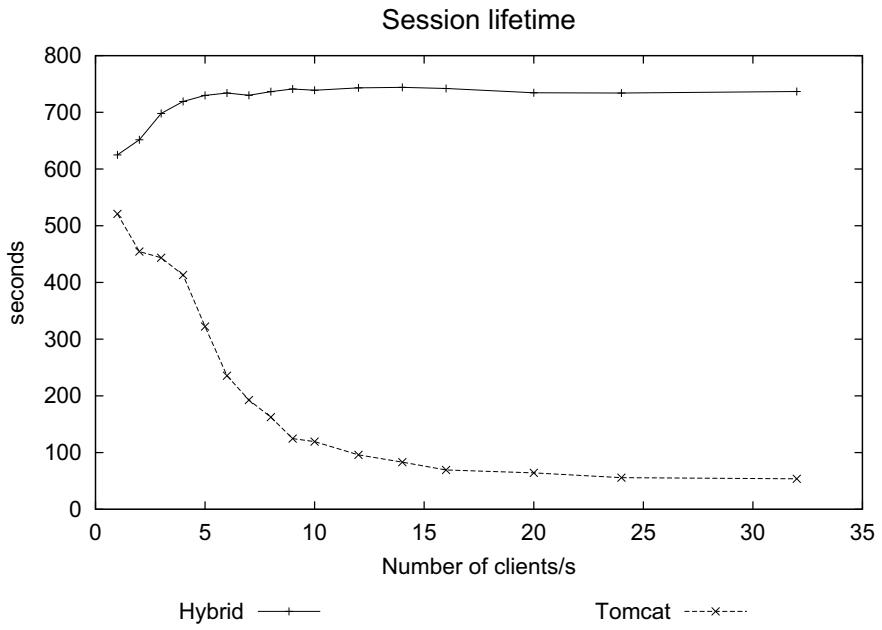


Figure 4.10 Lifetime comparison for the sessions completed successfully

by the longest sessions when the server is saturated under the multi-threaded architecture. In contrast, in the hybrid architecture the independence between the number of connected clients and the number of processing threads in the server allows this architecture to avoid the need for re-connections and their associated SSL handshakes, which allows long sessions to be completed without the difficulties associated with the multi-threaded architecture.

In addition to giving a higher chance to the long-lived user sessions to be completed, the hybrid architecture also increases the global average of sessions completed successfully, as can be seen in Figure 4.8. The original multi-threaded Tomcat server starts reducing the number of sessions completed successfully beyond its saturation point. This is caused by the higher number of sessions that are aborted by the clients when numerous re-connections are rejected by the server as it becomes increasingly overloaded. As the load increases in the server, the chance for a connection attempt to be accepted is reduced. As user sessions, for the multi-threaded model server, require a number of re-connections for each one, driving the server to higher loads increases the probability that clients consider the server unavailable if several connection attempts are rejected, which leads to a higher number of clients aborting their navigation sessions under these conditions. In contrast, the hybrid architecture avoids the need for client re-connections for a user session to be completed, which allows a higher number of them to finish successfully, independently of the system load.

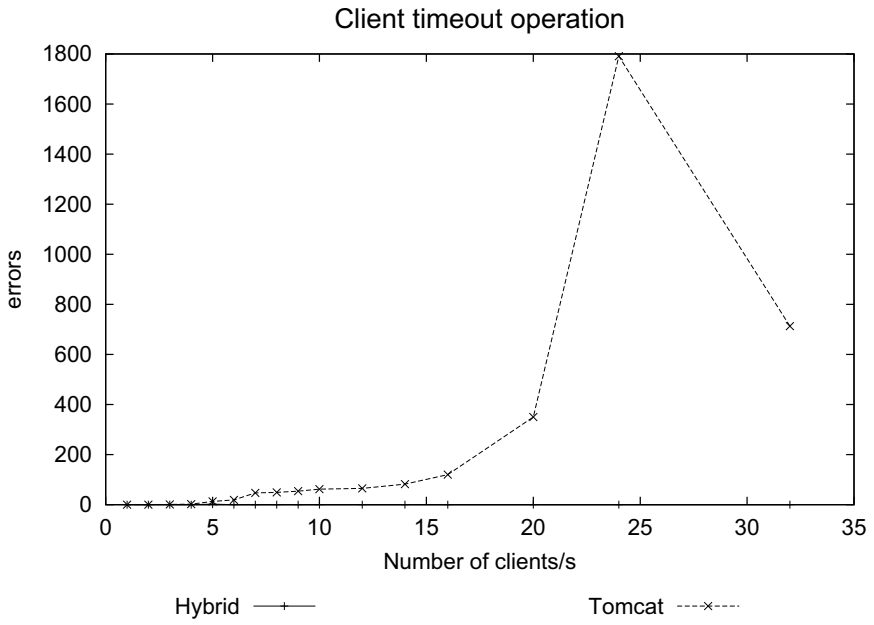


Figure 4.11 Number of client timeouts under a secure dynamic content workload

The benefits of the hybrid architecture presented above do not result in corresponding benefitsin the response time offered by it. As can be seen in Figure 4.9, the average response time obtained by the out-of-the-box Tomcat server is better than that obtained by the hybrid architecture, although the hybrid's performance is not unacceptable. A response time bounded to less than 250ms is more than acceptable and, moreover, it remains constant with the load. In our opinion a slight increase in the average response time obtained is more than acceptable considering the improvements in the server throughput obtained by the hybrid architecture.

Finally, another benefit of the hybrid architecture can be seen in Figure 4.11, where the number of requests sent to the server that do not produce a response after an acceptable period of time are shown. In the benchmark application this time is expressed as a timeout assimilated to the amount of time a human client would expect a reply from its web browser before considering a page request failed. For the experiments, this timeout value was set to 10 seconds. As can be observed, the hybrid architecture produces no client timeouts while the number of errors generated by the pure multi-threaded Tomcat server grows with the system load. This situation in the hybrid architecture is explained by the use of an overload control mechanism and also by the independence that exist between the number of connected clients to the server and the amount of concurrent server threads processing requests, which makes it possible to keep the latest requests in a low value and in this way reduce the contention caused by the multi-threaded architecture as well

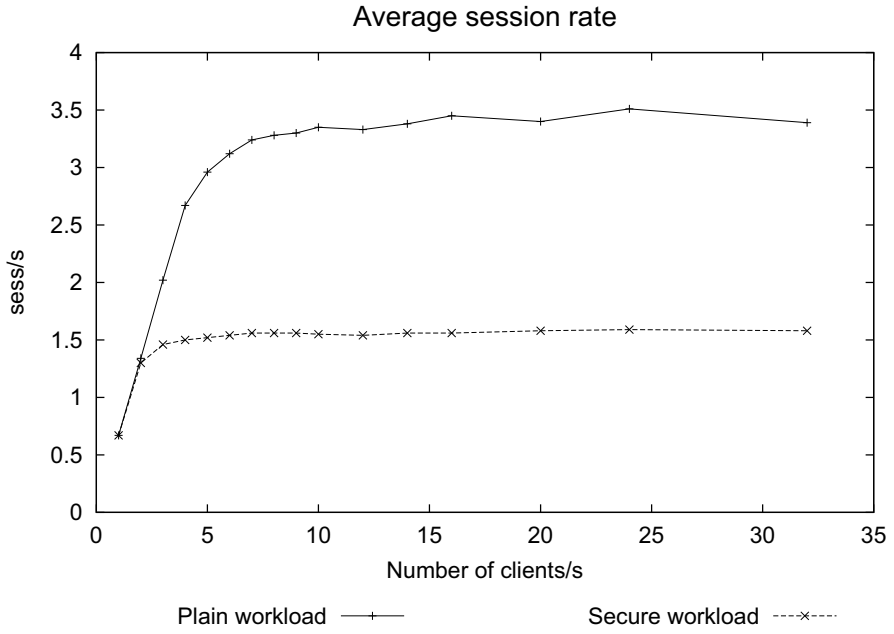


Figure 4.12 Session throughput for Hybrid server

as increasing the performance obtained by the server. It can also be seen that the number of timeout operation errors observed for the original Tomcat server decreases when the load level is driven beyond a certain point. This is because under that level of overload, the server starts rejecting connections because no server capacity remains available to process the volume of incoming connection attempts and less clients remain connected concurrently.

Secure vs. plain dynamic content

Once we had demostrated that for secure content workloads the proposed hybrid architecture provides a higher performance than the original pure multi-threaded Tomcat server in several ways, we wanted to measure the performance gap observed for both servers when subject to a plain workload and secure workload. This was done by rerunning the benchmark application with the HTTP protocol instead of HTTPS protocol used in the previous experiments. More results with plain workload are discussed in section 4.1.

Figures 4.12 and 4.13 highlight another remarkable effect of the hybrid architecture, namely that it reduces the impact on the performance caused by the introduction of security in the workload from a factor of 7 in the original Tomcat server to a factor of only 2 in the hybrid architecture as measured by the throughput in successfully completed user sessions.

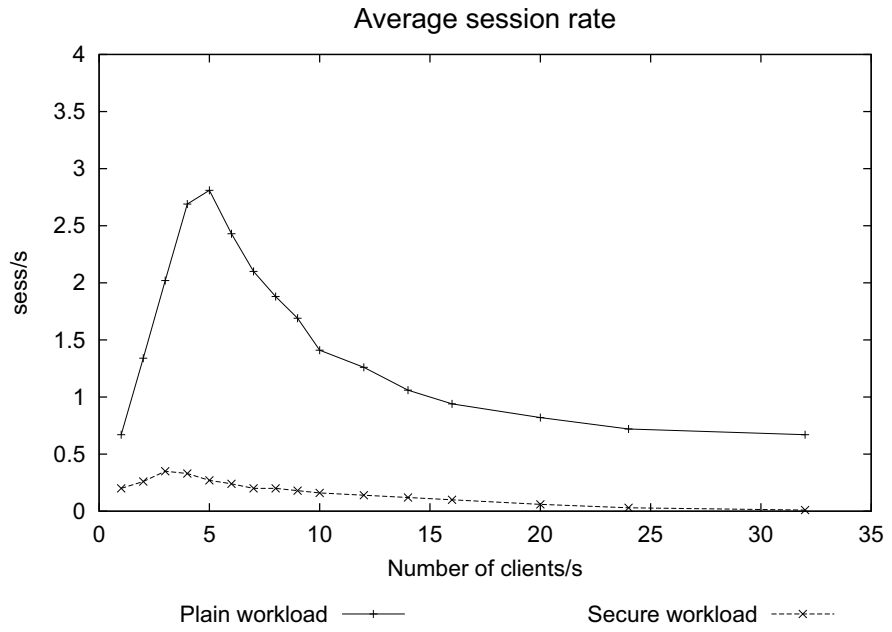


Figure 4.13 Session throughput for Tomcat server

4.2.5 Summary

In this section we have demostrated that the use of a hybrid web server architecture, which makes the best use of multi-threaded and event-driven architecture, considerably improves web server performance, especially under session-based workloads and even more when the workloads use encryption techniques. As has been demonstrated, the benefits of this architecture are especially noticeable in the throughput of the server, in terms of individual requests as well as for user sessions, particularly when the server is overloaded. The modified Tomcat server beats the original multi-threaded Tomcat in all the performance parameters studied and minimizes the impact of incorporating secure connections to a web application with respect to the out-of-the-box Tomcat.

4.3 Study of Web Server Configuration Complexity

Adequately setting up a multi-threaded web server is a challenging task because its performance is determined by a combination of configurable web server parameters and unsteady external factors such as the workload type, workload intensity and machine resources available. Usually administrators set up web server parameters like the keep-alive timeout and number of worker threads based on their experience and judgment, expecting that this configuration will perform well for the estimated uncontrollable factors. The nontrivial interaction between the configuration parameters of a multi-

threaded web server makes it difficult to properly tune the server for a given workload, but the burst nature of the Internet quickly change the uncontrollable factors and make it impossible to obtain an optimal configuration that will always perform well.

In this chapter we examine the complexity of optimally configuring a multi-threaded web server for different workloads by conductiong an exhaustive study of the interactions between the keep-alive timeout and the number of worker threads for a wide range of workloads. We also analyze the Hybrid web server architecture (multi-threaded and event-driven) as a feasible solution to simplify web server tuning and obtain the best performance for a wide range of workloads that can dynamically change in intensity and type. Finally, we compare the performance of the optimally tuned multi-threaded web server and the hybrid web server with different workloads to validate our assertions. We conclude from our study that the hybrid architecture clearly outperforms the multi-threaded one, not only in terms of performance, but also in terms of its tuning complexity and its adaptability over different workload types. In fact, from the results obtained, we conclude that the hybrid architecture is well suited to simplify the self configuration of complex application servers.

4.3.1 Introduction

The complexity and requirements of current web applications has grown enormously in the last few years. To accommodate this growth, the traditional web server which only returned static pages has evolved into an application server with a large number of functionalities and components. Good examples of this trend are the J2EE and .Net frameworks, which provide all the functionalities needed by a modern application; from security and reliability to database access. The down side of this evolution is that the complexity of configuring and managing this middleware (web server, application server, database ...) has grown correspondingly. The objective of this chapter is to present an alternative to the common multi-threaded architecture in this complex environment, with the aim of reducing the configuration and tuning complexity, as well as, improving the performance of the web servers under a wide range of realistic workload conditions.

The first part of the chapter gives a detailed description of how the interaction between the key configuration parameters of a multi-threaded web server and the workload characteristics affects the server's performance. From this a heuristic is devised to determine the optimal configuration parameters for the multi-threaded architecture. The second part of the chapter shows how the use of a hybrid architecture can dramatically reduce the complexity of setting up a web server and can also almost completly isolate the server's performance from changes in the workload intensity and type, which transforms

the optimal configuration of the server into a relatively trivial task. Finally, we compare the raw performance of the tuned multi-threaded architecture to the hybrid performance for three different application workloads.

The problem of configuring and tuning web servers and complex application server systems has been previously studied in literature. In [55] an analytical model is developed to provide a rigorous analysis of the effects of concurrency on web server performance. In [90] the authors formulate the problem of finding an optimal configuration for a given application as a black-box optimization problem. [78] provides an ad-hoc experimental methodology based on statistical techniques to identify key configuration parameters that have a strong impact on application server performance. Their results show that the number of threads and the value of the keep-alive timeout, are not only the key parameters of a web server, but also have a great influence on the performance of the entire application server. [71] presents an experimental methodology that randomly explores the configuration space of an application server and infers useful information based on multiple regression techniques. In [66] an algorithm to reduce the measurement time needed to explore a large configuration space and find a nearly best configuration of a web server is proposed. An interesting framework to automate the generation of configurations and tuning parameters for clusters of web servers is proposed in [93]. The focus of this work is the elimination of parameters misconfigurations when there are a cluster addition or removals of web servers. This work also studies the dependencies between the tuning parameters of a servers. All of the previous mentioned techniques provide good application server configurations for a given application workload but if a workload change occurs, the server may under-perform due to the nature of the multi-threaded architecture. [41] performs a fine-grained scalability analysis of a multi-threaded web server, but is manly focused on the impact of security on web server performance.

In [61] and [35], dynamic feedback-control based techniques are applied to optimize the number of threads and the keep-alive timeout in order to satisfy a given CPU and memory policy utilization. This approach can deal with variations in workload intensity but is unable to adapt to workload changes because it requires prior knowledge of resources which may be the bottlenecks.

Our goal is to demonstrate that the hybrid architecture can drastically reduce the configuration complexity of the web tier for a wide range of application workloads. In contrast to the multi-threaded architecture, the hybrid architecture can reduce the entire application server' tuning complexity and improve its adaptability to dynamic changes in workload intensity and type. To accomplish this objective we use a state of the art benchmark and an exhaustive experimental methodology to evaluate the behavior of a

wide range of web server configurations.

The rest of the chapter is organized as follows: Section 4.3.2 describes the environment and software used to perform the experimental tests. Section 4.3.3 presents the methodologies used to find the optimal configuration for both server architectures. Section 4.3.4 compares the performance of the optimally configured servers. Finally, Section 4.3.5 draws conclusions and discusses the future work.

4.3.2 Experimental environment

We performed all the tuning and performance measurements using the Tomcat web server and the SPECWeb2005 [9] benchmark. Tomcat is a popular web server that can be run stand-alone or embedded inside a J2EE server like JBoos or Geronimo. Like the Apache web server, Tomcat is a modular web server that supports the multi-threaded and the hybrid threading model, but does not support the pure event-driven one because this threading model does not fit well with the blocking semantics of the Servlet and JSP technologies used to implement the SPECWeb2005 applications benchmark. Usually, event-driven web servers are only used to serve static content, and redirect all the dynamic requests to a multi-threaded web server, increasing the configuration complexity of the system. Hence, the evaluation of the event-driven architecture is not feasible in the scope of this chapter.

SPECWeb2005 is a standard benchmark composed of three different application workloads. This benchmark is divided into three logical components that run on different IBM servers interconnected by a gigabit Ethernet switch. The first component is the distributed client emulator that runs on a group of OpenPower 720 servers. The second is the web server that runs on a JS21 blade with two Power970MPs, 8GB of RAM and two 60GB SCSI drives. Finally, the third component is the database emulator (BESIM) that runs on an OpenPower 710. All the machines run a 2.6 Linux system. In the next subsections, the SPECWeb2005 benchmark and Tomcat web server architecture are detailed.

SPECWeb2005

The large differences in security requirements and dynamic content of various web server workload types makes it impossible to synthesize them into a single representative workload. For this reason SPECWeb2005 [9] is composed of three workload applications that attempt to represent three different types of real world web applications.

The first is the Support workload, based on a web application for downloading patches

and this tests the ability to download large files over plain connections. The two principal characteristics of the Support application are the use of only plain connections and a large working set per client, so the benchmark tends to be I/O disk intensive. The second is the Banking workload that imitates an online banking web application. The principal characteristic of the Banking workload is the use of secure connections only, where all the transmitted data is encrypted / decrypted, and thus the benchmark becomes CPU intensive. The last is the E-commerce workload, based on an e-commerce web application. The principal characteristic of the E-commerce application is the mix of secure and plain connections to mimic the behavior of real e-commerce sites. Usually a client navigates through the application looking for a product over a plain connection and when they find it, they change to a secure connection in order to buy the product. The number of plain connections is usually greater than the secure ones and the working set per client is quite large, so the benchmark balances the CPU and the I/O disk usage, but tends to be more CPU intensive. The three workloads are based on requests for web pages and involve running a dynamic script on the server end and returning a dynamically created file, followed by requests for embedded static files.

The benchmark client emulator is based on a number of customers who concurrently navigate the application, switching from active to passive states and vice versa. A client in the active state performs one or more requests to the web server to simulate user activity, whereas a client in the passive state simulates a user think-time between active states. The time spent in passive states (or think-times) are distributed following a geometric distribution. For each workload type a Markov Chain is defined to model the client access pattern to the web application and a Zipf distribution is used to access each web application's working set.

SpecWeb2005 also has three other important features to mimic a real environment. Firstly, it includes the use of two connections for each client to perform parallel requests. Secondly, it simulates browser caching effects by using If-Modified-Since requests. Finally, the web server application working set size is proportional to the number of clients simulated.

Tomcat Web Server

Tomcat [39] is a stand-alone web server that supports Servlets and JSP. The standard Tomcat web server is composed of three major components: the Coyote connector, the Catalina container and the Jasper JSP compiler. The Coyote connector is in charge of accepting and managing the clients' connections and is implemented following a multi threaded architecture. The Catalina container and Jasper compiler share the multi-

threaded architecture of Coyote and provide the Servlet and JSP functionalities of the web server. A remarkable fact is that the same threads that manage the connections and parse the HTTP requests on Coyote also execute the code of the Catalina container and the JSP compiler. In the next subsections, the three versions of the Coyote connector currently available are described: the original Coyote that implements a multi-threaded architecture and the Hybrid and APR versions that implement the hybrid architecture. Each one of the connectors can be configured to manage plain connections as well as secure connections.

Multithreaded Connector

The original Tomcat implementation of the Coyote connector follows a pure multi-threaded approach to manage the client connections. For each incoming connection, one thread is assigned to serve the requests until the client closes it or a keep-alive timeout occurs. The two key configuration parameters are the number of threads and the keep-alive timeout value because both parameters determine the performance of the web server [78][35]. Coyote implements a best effort heuristic to avoid low performance due to mis-configured keep-alive timeouts and number of threads parameters. This heuristic dynamically reduces the keep-alive timeout when the worker threads get busy. We deactivate this feature to be able to study the combined parameter effects on the studied metrics without interference. As we will show later, this heuristic can not provide the best server performance for any of the workload types studied because the best configurations never have a short connection keep-alive timeout.

Hybrid Connector

The Hybrid connector is a modification of the original Coyote connector; although both connectors share most of the code such as HTTP request parsing code or the pool of threads. The Hybrid connector follows the architecture design explained in Chapter 4.1. The implementation of the hybrid connector is feasible thanks to the Non Blocking I/O (NIO) API [50] that appeared in Java version 1.4, and provides the select() call that permits the breaking of the thread-per-connection paradigm of the multi-threaded architecture. The Hybrid connector only has one configuration parameter: the number of worker threads, because the keep-alive timeout parameter is always set to infinite. For a more detailed description of the Hybrid connector implementation see [27].

APR Connector

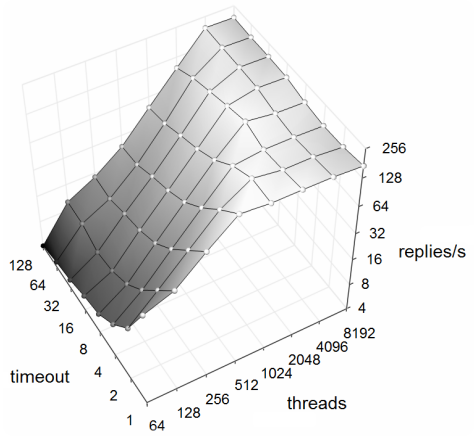
The APR and Hybrid connectors share the same hybrid architecture concept. The major difference between both connectors is that the Hybrid connector is based on the Java NIO API, whereas the APR connector is based on the native Apache Portable Runtime [2] and uses it to perform native select system calls, which are available in most operating systems. In this chapter, we do not attempt to evaluate the performance of the APR connector, as it is still under development. However, the preliminary tests of the APR connector show performance behavior similar to the Hybrid connector and do not show any significant performance improvements on our systems.

4.3.3 Methodology

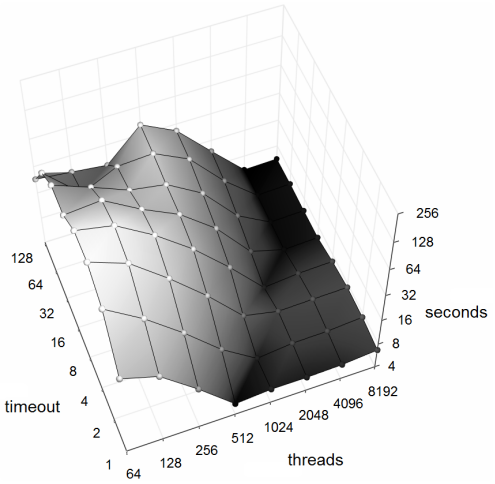
In this section, we first explain the methodology used to study the effect of the hybrid and multi-threaded configuration parameters on the performance for the three studied workload types: Support, E-commerce and Banking. With this information, we develop a heuristic to determine the best configuration parameters for the two architectures when given a workload type. To evaluate the tested configurations of both architectures, we focus on the throughput, response time and number of errors metrics which are results provided by the SPECweb2005 benchmark. To find the best configuration, we also calculate the throughput/response time metric, which shows the best configuration of a group with similar throughput. Each standard run of the Specweb2005 benchmark is composed of four phases: a thread ramp-up of 3 minutes, a server warm-up of 5 minutes, a steady-state (when the statistics are gathered) of 30 minutes and a thread ramp-down of 3 minutes.

Tuning the Multithreaded Architecture

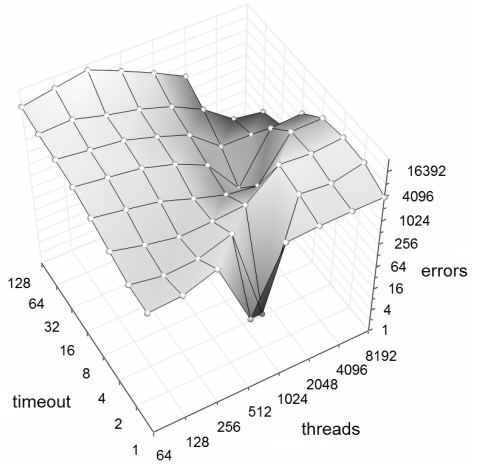
The multithreaded connector has two key parameters that influence performance; the keep-alive timeout and the number of threads. The difficulty in understanding their impact on performance is that their interaction also depends on the application's workload type. For this reason, for each workload type, we tested a large number of combinations of both parameters, which practically cover all the possible meaningful configurations, using a constant workload intensity of 2000 concurrent clients. The plot of the data obtained produces four 3D graphics for each workload type. Figures 4.14, 4.15 and 4.16 show the results obtained for the Support, Banking and E-commerce workload respectively. Notice that all the 3D graphics use a log scale on the three axes. The search space representing reasonable configurations for the three workloads are the product of the keep-alive timeout



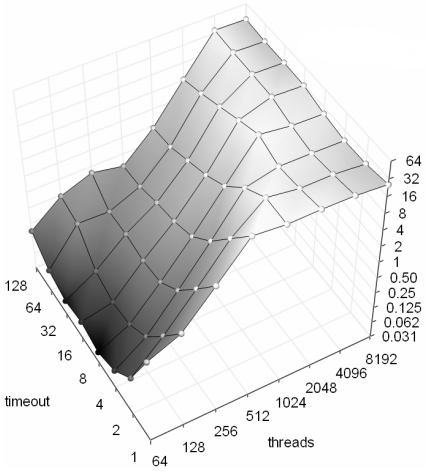
(a) Throughput



(b) Response Time



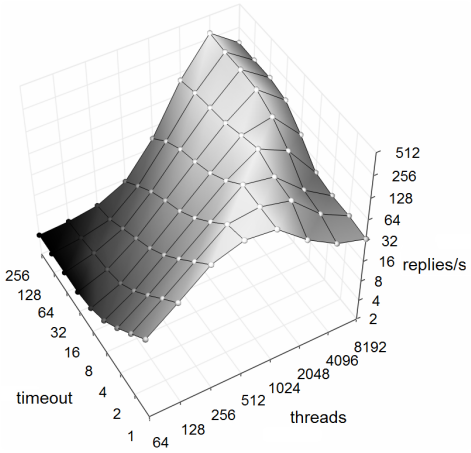
(c) Errors



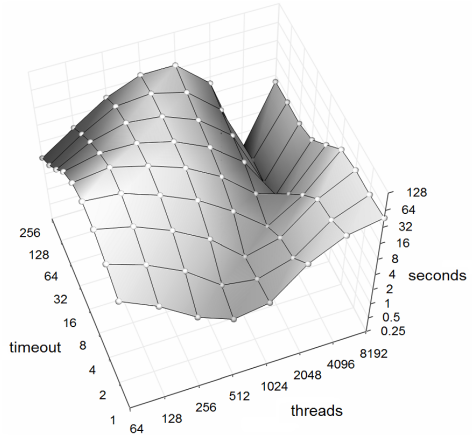
(d) Throughput / Response Time

Figure 4.14 Support Workload. Effect of Coyote Parameters. 2000 Concurrent Clients

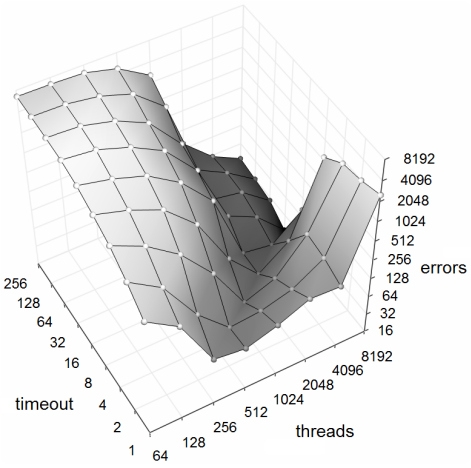
and the number of threads. The keep-alive timeout used in the tests varies from 1s to 128s for the Support workload and from 1s to 256s for the Banking and E-commerce workload. In both cases, the last value is equal to an infinite timeout because on the Support workload the maximum client think time allowed is 75s, while in the E-commerce and Banking workload it is 150s. The range of the number of threads used varies from 64 threads to 8192 threads. This includes a reasonable minimum and a maximum that doubles the number of connections that SPECWeb2005 establishes with 2000 concurrent clients (since each clients establishes two connections simultaneously, as browsers usually do with visited web servers). Examining the results of these tests, we determine the best multi-threaded architecture configuration (timeout and number of threads) for each workload type. The optimal configuration of each workload is be used to compare the



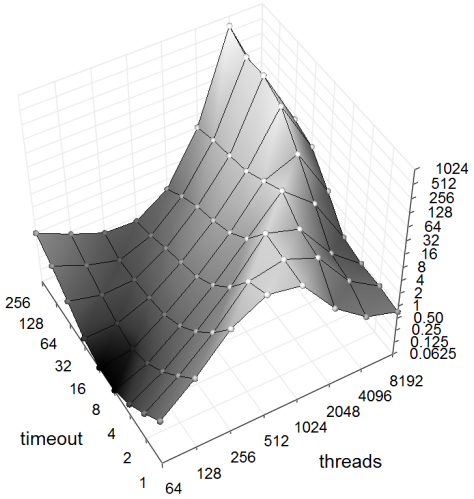
(a) Throughput



(b) Response Time



(c) Errors



(d) Throughput / Response Time

Figure 4.15 Bank Workload. Effect of Coyote Parameters. 2000 Concurrent Clients

performance between the two architectures in Section 4.3.4.

Support Workload

Figures 4.14a and 4.14d show that the best configuration for the Support workload is obtained when the server has at least one thread to serve each incoming connection (4096 or 8192) and a keep-alive timeout of 128s, which is equivalent to an infinite keep-alive timeout, meaning that the server never closes connections to inactive clients. The configurations with less than 512 threads, even with a low timeout, have the worst results in terms of throughput and especially with response time (as shown in Figure 4.14b) because there is not a sufficient number of available threads to serve all the clients. With less than one thread per connection (in this case less than 4000 threads because

we have 2000 concurrent clients) a large timeout also reduces performance and raises the response time. The number of errors shown in Figure 4.14c is always low for the optimal configuration of one thread per connection and an infinite keep-alive timeout. The number of errors metric accounts for connection timeouts, bad requests and data integrity errors. Almost all the errors in this benchmark are produced by connection timeouts due to an improper server configuration.

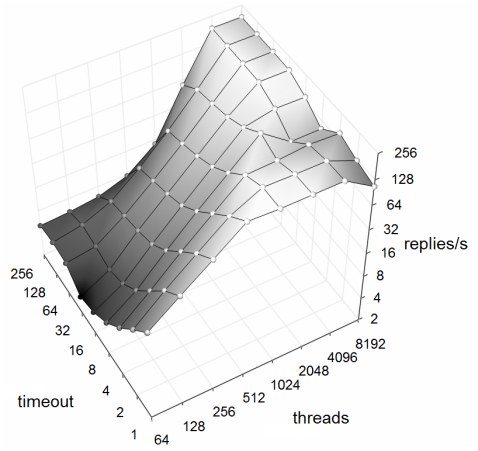
These results show what a big impact mis-configured parameters for a multi-threaded connector on the Support workload performance can have. From the results obtained, we can deduce that the best parameter configuration always has an infinite keep alive timeout and one thread per client connection.

Banking Workload

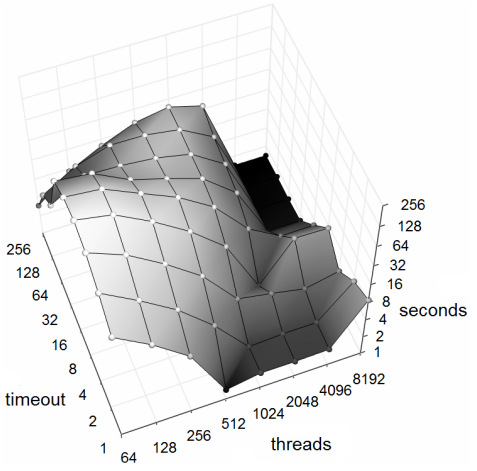
Figure 4.15a shows that no configuration with a keep-alive timeout of 8s or less performs well. This fact is explained by the impact that re-connections have when secure connections are used. The number of threads also impacts the performance of the multi-threaded connector. The best results are obtained with connections of only one thread per client and a large timeout, but with more than one thread per connection the performance of the multi-threaded connector drops because thread contention effects arise and the server is near its saturation point, as we discuss in the next section. The behavior of the Support and Banking workloads are very different for the multi-threaded connector because one is I/O intensive and the other CPU intensive. However the best configuration is the same for both workloads; one thread per connection and an infinite keep-alive timeout. While in the first workload, as we can see in Figures 4.14a and 4.14a, an increment in the number of optimal threads does not degrade the throughput or the response time. In the second case, the increment produces a remarkable reduction of the throughput and increases the response time as we can see in Figures 4.15a and 4.15b. This may be explained by the different bottlenecks that each workload type hits; in the first, the disk bandwidth and in the second, the CPU usage.

E-commerce Workload

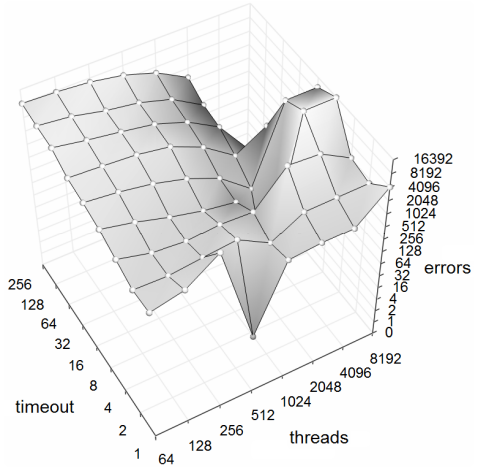
Figure 4.16 shows the four metrics for the studied E-commerce workload. We can deduce from these graphics, that the best configurations are also obtained using an infinite timeout and one thread per established connection. However, the set of configurations with one or more thread per connection and a keep alive timeout of at least 64 seconds obtains an equivalent performance. For this workload, as in the Support workload, when there is more than 512 threads we can obtain pretty good performance with a small



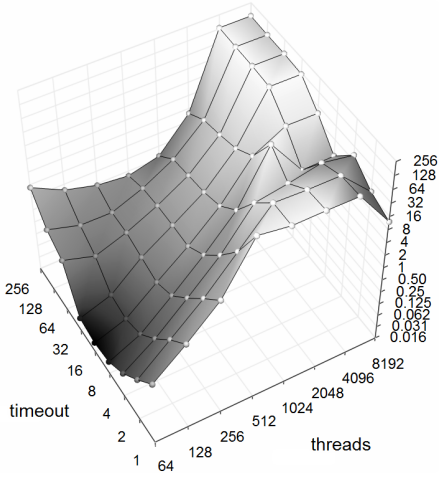
(a) Throughput



(b) Response Time



(c) Errors



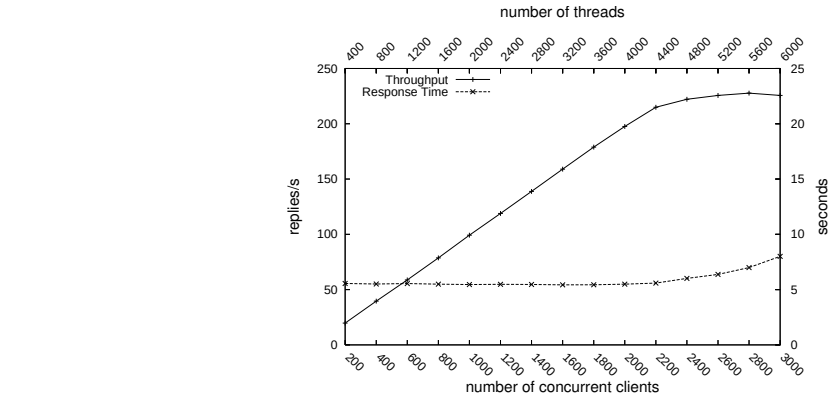
(d) Throughput / Response Time

Figure 4.16 E-commerce Workload. Effect of Coyote Parameters. 2000 Concurrent Clients

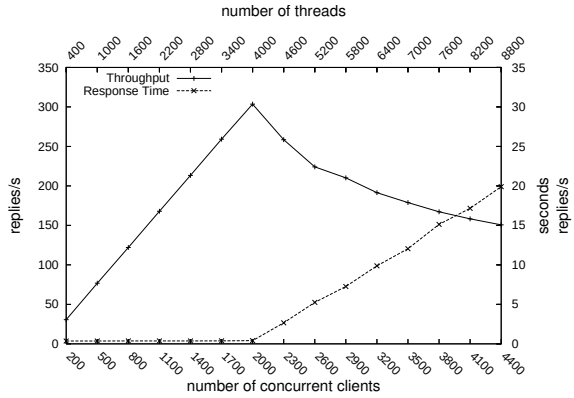
timeout, but within this class of configurations we can also find some configurations that reduce the performance. This type of behavior makes it more difficult to predict the performance of the multi-threaded architecture. Once again, we find that the best configuration is one thread per connection and an infinite keep-alive timeout.

Best Multithreaded Configuration

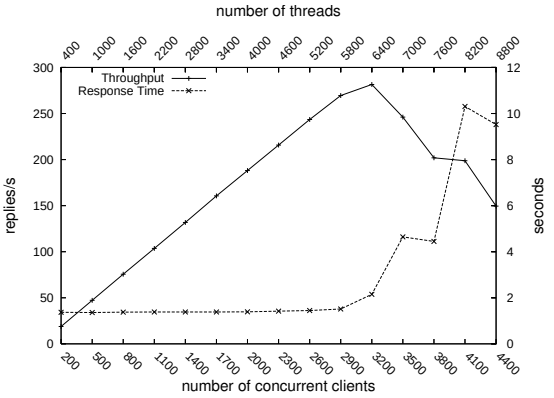
As we have seen in the last three sections, the best configuration for the multi-threaded connector is one thread per connection for the three workloads. This result is only valid for our systems because on other systems with more memory pressure or application contention, it can harm a configuration when it has a high number of threads. The only



(a) Support Workload



(b) Banking Workload



(c) Ecommerce Workload

Figure 4.17 Multithreaded Connector. Optimal Worker Threads Configuration

general conclusion that we can draw is that the optimal configuration of a multi-threaded web server crucially depends on the machine resources and workload type.

In Figure 4.17, we find the maximum workload intensity that the server can support with the one thread per connection configuration. The figures have a double horizontal and double vertical axis. The left vertical axis measures the throughput as requests per second while the right vertical axis shows the response time in seconds. The bottom horizontal axis shows the load as the number of concurrent clients and the upper horizontal axis shows the number of threads used to run the multi-threaded server. The SPECWeb benchmark establishes two connections for each client emulated so we need two threads for each concurrent client to use the optimal configuration of one thread per connection.

For the Support workload in Figure 4.17a we see that the best results are obtained with 2400 concurrent clients and 4800 threads, because after this point the response time starts to increase, but the throughput remains constant. In Figure 4.17b, we can appreciate that the best performance for the Banking workload is reached with exactly 4000 threads because from this point on the throughput starts to degrade quickly. The

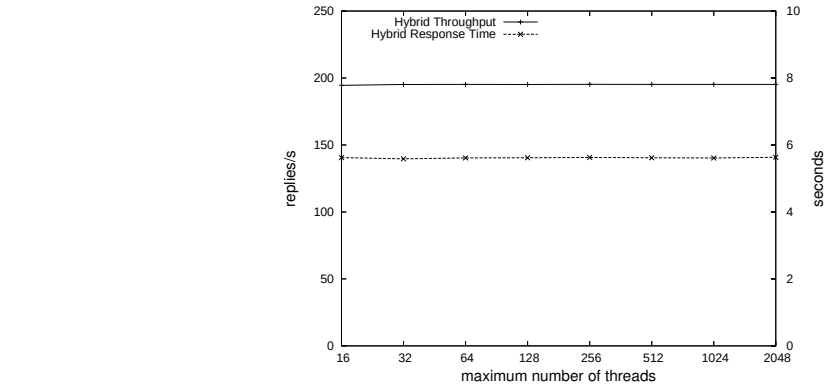
same behavior can be seen in Figure 4.17c for the E-commerce workload that reaches its maximum throughput for 3200 concurrent clients with a configuration of 6400 threads. The different behavior observed between the Banking and E-commerce workloads and the Support workload can be explained by the bottlenecks that each workload type hits; in the first, the CPU usage and in the second, the disk bandwidth.

Tuning the Hybrid Architecture

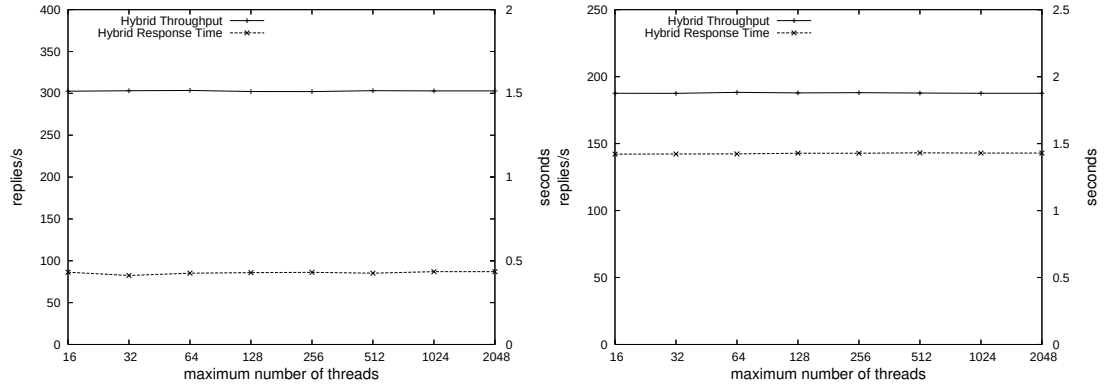
The hybrid connector only has one configuration parameter, which is the maximum number of concurrent threads existing in the thread pool. The connection inactivity timeout can be ignored because it is always set to infinite to avoid the overhead of client's re-connections. This fact dramatically simplifies the number of experiments needed to determine the hybrid connector's performance behavior. This is depicted in Figures 4.18a, 4.18b and 4.18c which show the throughput and response time for the Support, Banking and E-commerce workloads respectively. These results are obtained with a constant workload intensity of 2000 concurrent clients as in Section 4.3.3. With this reasonable load, which does not overload our server, we measure the performance metrics while varying the number of worker threads from a minimum of 16 to a maximum of 2048. Notice that the three graphics in Figure 4.18 have a logarithmic horizontal axis. With the results obtained, we determine the best configuration for each workload type, and use it to evaluate the performance of the Hybrid connector in Section 4.3.4.

Support, Banking and E-commerce Workload

Figure 4.18 shows that varying the number of threads between 16 and 2048 has no effect on the performance of the server. The throughput and the response time remain constant independently of the number of threads used or the workload type. The error metric is not shown because it is zero for all the experiments. This robust behavior can be explained by the hybrid architecture design, which only needs a small number of threads to serve a large number of clients. In fact, we are not setting the number of concurrent threads, but the maximum number of threads that can be used. This fact explains why the server's performance does not degrade when configured with 2048 threads because the server only uses the minimum number of threads necessary to handle its current workload intensity. The only way that the server can perform badly is if it is configured with a very low number of threads. Remarkably, the use of secure connections in the Banking and E-commerce workloads does not change the hybrid connectors' performance behavior because the infinite timeout used avoids the needs for client re-connections and reduces the number of SSL handshakes. Although the Support, Banking and E-commerce



(a) Support Workload



(b) Banking Workload

(c) E-commerce Workload

Figure 4.18 Hybrid Connector. Optimal Worker Threads Configuration

workloads are completely different, the performance behavior of the hybrid server is the same on the three workloads as we notice when comparing Figures 4.18a, 4.18b and 4.18c. >From the results shown in Figure 4.18, it can be deduced that with 16 threads, the server can manage at least 2000 concurrent clients. Hence, at least 64 threads will be used for the performance comparison of section 4.3.4, where the workload intensity reaches 4400 concurrent clients for the Banking and E-commerce workload. The results obtained show the simplicity of configuring the Hybrid architecture because we only have one parameter to configure and this parameter, the number of threads, always performs optimally if it is larger than a small threshold.

Connector Configuration Conclusion

From the studied workloads and the results obtained, we can see the qualitative difference between the complexity of configuring both architectures. While the hybrid connector performs well with any reasonable configuration on each workload type, a big effort is needed to find a good custom configuration for the multi-threaded connector.

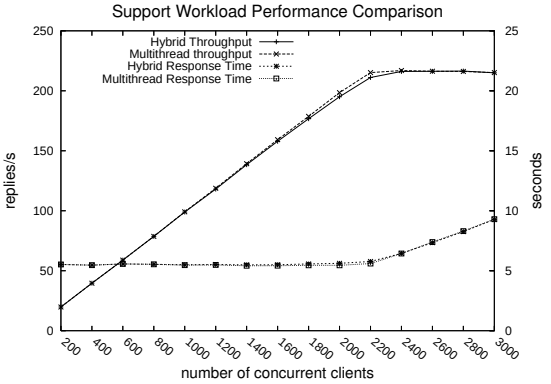
First, we need to understand the interaction of the keep-alive timeout and the number of threads for a given workload type and hardware, in order to find the optimal configuration. On our hardware, we find that the optimal Multithreaded connector configuration always has the form of one thread per connection and an infinite keep-alive timeout. However, this result is only valid for our machine, as with one thread per connection the memory requirements may lower the performance on other machines which have more memory pressure. Moreover, after determining the best parameter configuration of the Multithreaded connector for a given workload type, we need to find the highest throughput that the specific server can achieve by varying the number of threads. This is an expensive process that must be applied each time the workload type or the hardware changes. In others environments like web farming, when one server hosts different types of applications, this approach is not feasible because the optimal configuration for one application may be inadequate for the rest of the applications. In this scenario the Hybrid architecture is the only feasible way to obtain optimal performance from the hardware.

4.3.4 Performance results

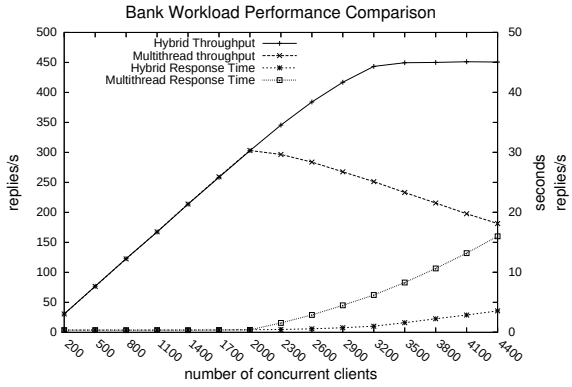
In this section we compare the raw performance of the optimally configured hybrid and multi-threaded connectors for each workload type. The metrics measured are the throughput and the response time, while varying the workload intensity, to capture the connectors' behavior. The hybrid connector is configured with a maximum of 64 threads for all the workload types, while the multi-threaded connector uses its optimal configuration for each workload.

Support Workload Performance

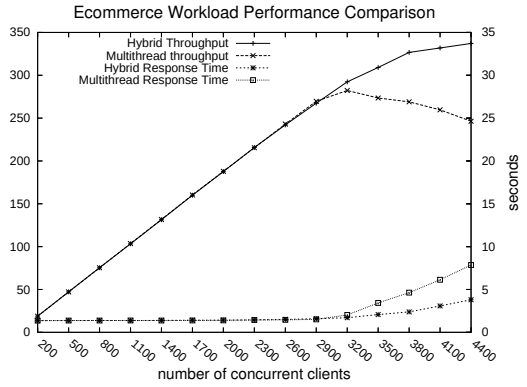
In this experiment, the workload intensity ranges from 200 to 3000 concurrent clients in increments of 200. The multi-threaded connector is configured with 4800 threads and an infinite timeout as we have determined in the previous section from Figures 4.14 and 4.17a. We can see in Figure 4.19a that the performance of the Hybrid and Multithreaded connectors are equal in both throughput and response time. This fact can be explained by taking into account that the workload is I/O disk intensive, so the overhead of a large number of threads used by the multi-threaded connector is hidden by the disk bottleneck. We measure that at the peak performance point the CPU utilization is under 60% and 50% for the multi-threaded and hybrid connector respectively.



(a) Support Workload



(b) Banking Workload



(c) E-commerce Workload

Figure 4.19 Multithreaded and Hybrid Performance Comparison

Bank Workload Performance

In this experiment, the workload intensity ranges from 200 to 4400 concurrent clients in increments of 300. The multi-threaded connector is configured with 4000 threads and an infinite timeout as we have determined this configuration to be the optimal from the experiments showed in the Figures 4.15 and 4.17b. Figure 4.19b shows the throughput and response time of both connectors. As we can see, with up to 2300 concurrent clients, the two connectors show the same performance, but from this point on, the multi-threaded connector's throughput starts to degrade. On the other hand, the hybrid connector linearly increases its throughput until it reaches 3200 concurrent clients where its performance stabilizes until the last point. The peak throughput of the Hybrid Connector is 50% higher than the Multithreaded Connector. The difference in raw performance can be explained by the large number of threads used by the multi-threaded connector which produces high contention and CPU usage due to the pressure on the scheduling, memory management and garbage collection components of the Java virtual machine.

E-commerce Workload Performance

In this experiment, the workload intensity ranges from 200 to 4400 concurrent clients in increments of 300. The multi-threaded connector is configured with 6400 threads and an infinite timeout as we have determined this to be the optimal configuration from the experiments showed in Figures 4.16 and 4.17c of the previous section. The performance results obtained are similar to the Bank workload type as shown in Figures 4.19b and 4.19c because the E-commerce workloads tend to be more CPU intensive than I/O intensive. In this case, the peak throughput is 30% higher for the Hybrid Connector. We measured the garbage collection time for both connectors and it is from 5 to 10 times larger in the multi-threaded one. This fact explains the difference in performance between the hybrid and multi-threaded architecture in the CPU intensive workloads.

4.3.5 Summary

We have studied the problem of configuring a web server for optimal performance. From our results we have demonstrated the inherent difficulty of optimally configuring a multi-threaded web server due to the interaction between its internal configuration parameters and the external workload characteristics. We have shown how the hybrid architecture can dramatically reduce the configuration complexity and increase the dynamic web server's adaptability to workload intensity and workload type changes thus making it a good solution for environments like web farms, where the servers need to deal with different types of workloads at the same time. Furthermore, the hybrid connector shows better performance than the multi-threaded one on the two CPU intensive workloads (50% and 30% throughput improvement for the Banking and E-commerce workloads respectively) and the same performance on the I/O disk intensive workload.

We suggest that further research should focus on studying the benefits that the hybrid architecture can contribute to a complex environment like J2EE or .Net, and the synergies that can be produced by using techniques for self-configuring in these environments, such as queueing theory and feedback control theory.

4.4 Disk and Memory bottlenecks

In Section 4.3.4, we evaluated the performance of the hybrid and multi-threaded web servers for three different workloads. In the two CPU intensive benchmarks the better connection management of the hybrid architecture provides a noticeable improvement on the overall web server performance. In contrast, in the I/O intensive workload both

web servers have identical performance and neither of them is capable of making the most of the underutilized CPU resources. In this workload the primary bottleneck is the disk bandwidth, which can be alleviated by improving the I/O subsystem or by adding more memory to the system in order to cache the most frequent request to disk. Both options can significantly increase the cost of the commodity hardware used to run current web servers, while the CPU resources are underutilized. Moreover, if the web workload changes this expensive disk or memory resources will be underutilized. For this reason, in the next chapter we present a software technique that uses main memory compression techniques to make the most of CPU underutilized resources and to improve the performance of bandwidth and memory bounded applications, without increasing the cost of the commodity hardware.

4.5 Summary

In this chapter we have presented and evaluated the Hybrid web server architecture. This Hybrid architecture mixes concepts of both the multi-threaded and the event-driven architecture. As we have seen in Section 4.1 and 4.2 the Hybrid architecture outperforms the multi-threaded architecture in all the evaluated workloads. Moreover, Section 4.3.5 shows how the Hybrid architecture dramatically reduce the configuration complexity of the whole web server system. As opposed to the event-driven architecture, the Hybrid one can be easily integrated with current multi-threaded web server and legacy information systems, thanks to its limited use of non-blocking operations in the client connection management. This fact has facilitated the rapid adoption and implementation of the key concepts behind the Hybrid architecture in well known web servers like Tomcat [4] [3], Apache Httpd [1] or Glassfish [10].

Chapter 5

Main Memory Compression

Current web servers are highly multi-threaded applications whose scalability benefits from the current multi-core/multiprocessor trend. However, some workloads can not capitalize on this because their performance is limited by the available memory and/or the disk bandwidth, which prevents the server from taking advantage of the computing resources provided by the system. To solve this problem we propose the use of main memory compression techniques to increment the available memory and mitigate the disk bandwidth problem, allowing the web server to improve its use of CPU system resources.

The objective of main memory compression techniques is to reduce the in-memory data size to virtually enlarge the available memory on the system. The main benefit of this technique is the reduction of slow disk I/O operations, thus improving data access latency and saving disk I/O bandwidth. On the other hand, its main drawback is the large amount of CPU power needed by the computationally expensive compression algorithms, that make it unsuitable for medium to large CPU intensive applications.

With the proliferation of multi-core systems, the amount of available CPU power is growing fast. In this scenario, the number of applications that can transparently benefit from memory compression can be expanded. Now, not only, single threaded applications, bounded by disk latencies, but also multi-threaded ones, bounded by the disk bandwidth can benefit from memory compression techniques.

In this chapter we implement in the Linux OS a full SMP capable main memory compression subsystem to increase the performance of a web server running the SPECWeb2005 benchmark. Although main memory compression is not a new technique per-se, its use in a multi-core environment running heavily multi-threaded applications like a web server introduces new challenges in the technique, such as scalability issues and the trade-off between the compressed memory size and the computational power required to achieve it. The evaluation of our implementation shows promising results such as a 30% web server throughput improvement and a 70% reduction in the disk bandwidth usage. Finally, we extend our main memory compression system to run on the heterogeneous Cell processor, where the compression and decompression tasks are offloaded to specialized coprocessors. With our implementation we prove the feasibility of completely offloading the computational expensive compression task to specialized processors, allowing the main processors to focus on the execution of user applications, thus increasing the range of applications that can benefit from it.

5.1 Introduction

Generally speaking, compressed memory systems are based on the reservation of some physical memory to store compressed data, virtually increasing the amount of memory available to the applications. This extra memory reduces the number of accesses to the disk and allows the execution of applications with larger working sets without trashing. However, the benefits of the compressed memory systems greatly depends on both the application access pattern and the data compression ratio, as well as, the ratio of compressed/uncompressed memory configured.

Previous work has exploited the compressed memory systems to accelerate the execution of single threaded applications with a large working set, exchanging high latency disk access for faster compressed memory access. This approach uses the idle times that this type of application usually spends accessing the disk to perform the decompression of the data requested. In contrast, we are interested in investigating the benefits that compressed memory systems can contribute to disk I/O bandwidth bound applications like a web server running the SPECWeb Support workload. In this case, the problem is that the web application is bounded by the available I/O bandwidth of the disk. A compressed memory system can mitigate this problem by providing more available memory to cache disk content in memory, thus reducing the number of accesses to the disk and the effective disk I/O bandwidth needed. A major challenge with this approach is the large amount of CPU power needed to provide the adequate bandwidth between the non compressed memory and the compressed one and vice versa. However, this CPU power is now more easily available with the proliferation of multi-core and multiprocessor systems which can be utilized for this purpose.

In summary, this chapter is focuses on ways of improving the performance of a highly multi-threaded web server running a disk-bounded web application. To this end, we implemented the first full SMP capable Compressed Page Cache (CPC) in the Linux OS that make the most of multi-core/multiprocessor architectures. To accomplish our objective we solved two challenging scalability issues. Firstly, we implemented our CPC on a multiprocessor Linux system that can scale in a highly threaded environment like that provided by a web server running the SPECWeb2005 benchmark, and secondly, we fructuously used a large fraction of the physical memory to store compressed data without running out of memory. As we will show, our novel CPC proposal is able to work with optimal performance when up to 85% of the memory is dedicated to store compressed data (in contrast to previous proposals that have been evaluated with a much smaller fraction of compressed memory, e.g. 10-20% in [83]).

The rest of the chapter is organized as follows: Section 5.2 presents the related work.

Sections 5.3 and 5.4 introduce the design goals and implementation of our prototype respectively. Section 5.5 explains our experimental environment and the performance results of our prototype. Section 5.6 describe the extensions required to run our prototype on the Cell heterogeneous processor. Finally section 5.7 draws the main conclusions of this chapter.

5.2 Related Work

Web servers are a well studied subject in the literature; issues such as performance [27], scalability [41], overload [42] or security [19] have been widely discussed but, to the best of our knowledge, the improvement of web server performance with memory compression techniques has not been studied before. The rationale behind using memory compression techniques to improve web servers' performance is based on the fact that the bottleneck of a web server for some workloads is the disk bandwidth and we can mitigate it with more memory, at the expense of CPU cycles, to perform the data compression. With the expansion of multi-core and multiprocessor systems, the CPU resources needed to make this technique feasible are currently available.

Memory Compression implementations can be classified as either hardware or software approaches, with their implied advantages and drawbacks. From the point of view of performance, the hardware approach is the best, mainly because of the utilization of custom hardware specialized for this purpose. In contrast software approaches make the most of underutilized CPU resources to provide a performance improvement for a typically more modest system. In this chapter we focus on software based approaches but, for the sake of completeness, we will also perform a brief review of the most relevant hardware approaches.

From the first group we can cite [91], [14], [38] which are all based on simulation techniques. The results of these studies show noticeable improvements in the amount of available main memory and the system performance, but they all need specialized hardware not available in current systems. To the best of our knowledge, the only main memory compression hardware approach implemented is the IBM MTX technology [8]. In [82] and [5] the MTX technology and the operating system modifications needed to run on top of the compression hardware are described. In [13] a performance evaluation with promising results over different workloads is presented. However, ultimately this project was not a success, probably because of the cost of the specialized hardware versus the cost of memory chips.

Software approaches implement compression techniques on top of commodity hard-

ware without any special support. In this section we review the most relevant works in this area. The first memory compression proposal, by Wilson [88], attempted to improve system performance by reducing the latency associated with disk access. In [37] Douglis implemented the first adaptive memory compression scheme in Spirit OS, based on a global LRU that can improve or decrease the performance of the system depending on the workload characteristics. Kaplan et al. [57] studies the adaptive memory compression scheme proposed by Douglis through simulation and found that the proposed scheme was partly at fault for some workloads. Kaplan also contributed the WK family of compression algorithms designed for in-memory data representations rather than file data. Finally he proposed a method to determine how much memory should be compressed during a phase of program execution by performing an online cost/benefit analysis, based on recent program behavior statistics.

In [30] Cervera et al. implemented in the Linux OS a compressed swapping mechanism to reduce the number of times the system had to access the swap device. Although the amount of compressed swap memory used was rather small, they observed a noticeable improvement of system performance. This is the first study to swap out pages to the swap device in compressed form, virtually increasing its capacity. Freedman et al. [6] applies memory compression techniques to reduce the power consumption and to improve the speed of embedded systems. Their compressed cache implementation was based on a log-structured circular buffer that allowed the compressed cache area to be dynamically resized. They estimated that compressed memory improves the disk access in both power efficiency and speed by 1-2 orders of magnitude. In [74] Roy et al. also proposes using compressed memory in order to hide the large latencies associated with disk access. They claimed that the optimal fraction of memory that should be reserved for compression lies at around 25% across a wide range of application types but they failed to provide a more general approach to set the memory compression size. In [34] Rodrigo reevaluated the use of adaptive compressed caching to improve the system performance. The main idea behind their proposal remains and it is to reduce the amount of disk accesses to improve the data access latency. Their contribution was a new adaptability policy that adjusts the compressed cache size on-the-fly based on the recent program behavior. They implemented the compressed cache in the Linux kernel and it was the first to provide file backed memory compression as well as swap based memory compression. They used the WKdm specialized compression algorithm to compress swap based pages and the LZO generic algorithm to compress file based memory pages. Their implementation provided noticeable improvements for a wide range of workloads and minimum overhead for the rest. Tuduce [83] proposed a new heuristic to dynamically

determine the compressed cache size with the objective of keeping all the application's working set in memory. Their results showed increases in performance by a factor of 1.3 to 55 times in three single threaded applications. Finally, in [73] Nitin Gupta ported Rodrigo's implementation of the compressed cache from kernel 2.4 to kernel 2.6 under the Google Summer of Code program for the OLPC project [65]. The work was based on the work and ideas of Kaplan, Rodrigo and Irina and the main objective was to increase the tiny memory available on the OLPC laptops.

A number of multi-threaded applications with a large working set, like a web server running the SPECWeb Support application, have a bottleneck in the I/O bandwidth rather than in the I/O latency, because their multi-threaded nature hides the latency. None of these studies reviewed above studied memory compression from the point of view of disk I/O bandwidth. We focus our discussion on the multi-core and multiprocessors systems as today they are standard commodity hardware. To the best of our knowledge, our implementation is the first to take full advantage of the new multi-core and multiprocessor system'. Another remarkable characteristic is that it is highly scalable in the amount of RAM that can be used to store compressed data (up to 85% of the physical RAM). Finally, our implementation is the first to allow the use of specialized processors to offload compression and decompression tasks.

In this chapter we do not compare our implementation with previous memory compression proposals (such as [34] and [83]) because it would produce equivalent results for the workloads that have already been studied. Instead, we focus on the evaluation of a multi-threaded web server with a disk bandwidth-bound workload on a multiprocessor environment, which is not supported by early implementations of main memory compression techniques. We expect similar results for other workloads that are also bounded by the disk bandwidth.

5.3 Compressed Page Cache Design

We chose the Linux operating system to implement our compressed memory subsystem and it basically augments the unified page cache to store compressed pages. We focus our discussion on the 2.6.24 kernel as it is the most recent stable version. In the following section, we briefly describe the overall Linux memory management subsystem and the relevant kernel algorithms needed to implement our compressed page cache (CPC). In the last subsection we explain our CPC design goals and the implementation details.

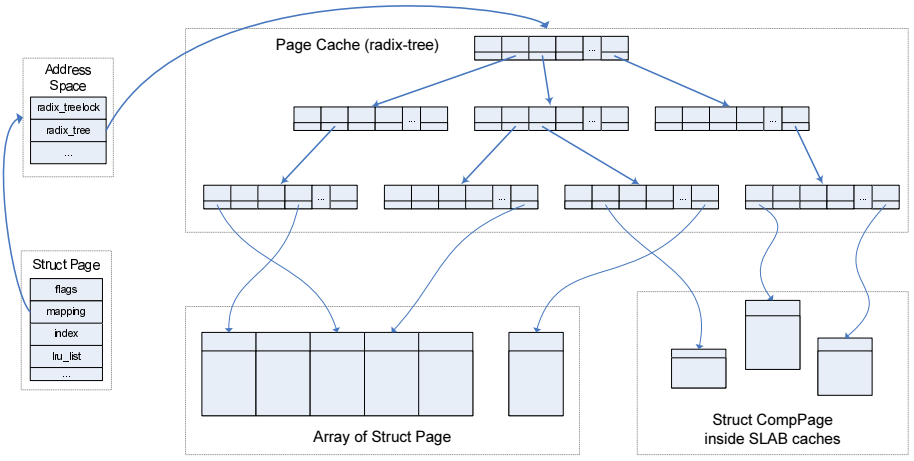


Figure 5.1 Unified Page Cache Diagram

Linux memory management

Linux memory management is developed around the core concept of the page frame. All the memory available on the system is divided in to page frames of the same size (usually 4Kb), but their size can be larger in some architectures like PowerPC or Itanium. The page frame is the smallest unit of work to manage the system memory. The content of a page frame changes dynamically depending on the needs of the system. Inside the kernel, there is an array that contains one "struct page" for each page frame of memory present in the system. Each struct page contains meta-data about a page frame and is implicitly associated with each page frame by its position in the array. Generally speaking, one page frame may contain three types of data: anonymous pages, file backed pages and private kernel pages. The anonymous pages contain data dynamically allocated from user space programs and can be swapped out under memory pressure if a swap device exists. File backed pages contain data that come from filesystem I/O operations. Finally, private kernel pages are used and managed by the kernel and device driver code for private purposes and can not be swapped out. One example of a private purpose is the SLAB allocator that provides caches of different sizes to the kernel code. The anonymous pages and the file backed pages form the unified page cache. All the I/O operations take place through this unified cache. When a page fault occurs or an I/O operation is required, the kernel always checks the unified page cache to find the requested data. If the data is not in the page cache, the kernel adds a new page frame to the page cache and performs the required I/O operation so that the page cache is always up to date.

The main data structure behind the unified page cache is a radix-tree that works as an efficient dictionary, mapping keys with struct pages - which uniquely identifies a page frames. Each key is formed with a mapping plus an offset that identifies whether a page

frame belongs to a file or a swap device. We can see a simplified diagram illustrating the radix-tree in Figure 5.1. All the struct pages of the unified page cache are linked together with a linked list to track their activity with a LRU like algorithm. When the system is under memory pressure, the kernel tries to free batches of pages from the tail of the LRU list until enough memory is available. In this process, the anonymous pages are usually swapped out, while the file backed pages are synchronized with the filesystem if needed, and then discarded.

5.4 Design Goals and Implementation

The objective of our design is to extend the unified page cache of Linux and provide a high performance compressed page cache (CPC) that can fully exploit the power of current multiprocessor systems. We also want to be able to use a large fraction of the physical memory to store compressed data because our web server workload (SPECWeb Support) has a huge working set, many times larger than the available memory. Another design objective is to minimize the number of changes made to the Linux kernel, avoiding the addition of complex algorithms or data structures.

The main idea behind the CPC is to modify the current unified page cache so as to also contain compressed page frames. Each compressed page frame has an augmented struct page called struct cpage, which is dynamically created to manage its content. This struct cpage is an extension of the standard struct page but contains additional information about the location and size of the compressed page frame. We mark one unused bit of the flags field in order to identify the pages that are currently compressed. The CPC modifies the normal flow of the pages in the LRU list in order to compress some of them to increase the size of the page cache. When a page frame is ready to be safely freed from the LRU, the page frame is compressed in a per-cpu temporary buffer. Then, the original page frame is freed and, if the compression is successful (i.e. the size of the compressed page frame is less than the original page frame size), the CPC allocates a number of buffers from the SLAB allocator and splits the content of the temporary buffer into these buffers, as we can see in Figure 5.2. The content of the compressed page frame is recursively divided in blocks of 2^n bytes and then copied to caches of the same size allocated on top of the SLAB. The number of blocks and their minimum size is predefined. As a result, the last block is larger than the remaining page frame bytes. The size of the last block is a tradeoff between a space overhead and a time overhead. The smaller the minimum predefined size, the larger the number of blocks needed to allocate a given compressed frame. In contrast, the larger the minimum predefined block size, the smaller the number of blocks needed

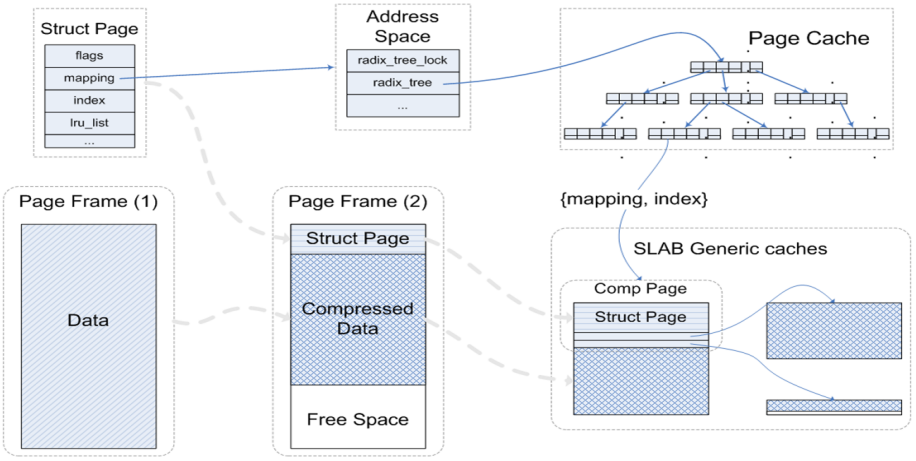


Figure 5.2 Diagram of Page Frame Compression

but the larger internal fragmentation, thus the space overhead. We fixed the maximum number of blocks to seven and the smallest block size to 32 bytes. As an example, if the size of the compressed data in the temporary buffer is 2788 bytes, we split it into three buffers of the following sizes: 2048, 512 and 256. To manage the new compressed page frame we dynamically allocate a struct cpage that contains a pointer to each SLAB buffer, which allows us to manage the new compressed page frame as a normal page frame because it is an extension of a struct page. The new allocated struct cpage is inserted on the head of the LRU list so that it can be reclaimed if it is not used in a period of time by the normal reclamation code of the Linux kernel.

When the page cache is queried, the CPC checks to see if the returned page is compressed or not. If it is compressed, the CPC proceeds to allocate a new page frame, and decompresses the content of the compressed page frame, releasing all the buffers to the SLAB and returning the new page frame. If there is no memory available to decompress the compressed page frame, then it is discarded and its memory returned to the SLAB. The query to the CPC will return null and the kernel will take the appropriate actions to read the data from the swap device or the filesystem. The main advantage of this storage system is its simplicity and robustness, because it is based on a fundamental Linux subsystem that is constantly being improved, namely the SLAB allocator. Moreover, building our compressed page cache on top of the current SLAB allocator allows us to dynamically adapt the space required to store the compressed page frames.

In summary, we capture a page that is close to being discarded from the page cache, compress it, split its content on top of the SLAB allocated buffers, and update its reference in the radix-tree to point to the new compressed page. The original page frame is discarded and the new cpage is inserted at the head of the LRU list. If it is not referenced in a period

of time, then it is discarded by the kernel reclamation code. When a lookup on the page cache returns a compressed page, we allocate a new page frame and fill it up with the decompressed data and the SLAB buffers that contain the old data are returned to the SLAB allocator. If the allocation of the new frame fails, the compressed page frame is discarded and a null value is returned, so the kernel takes the appropriate actions to read the required data from the filesystem or swap device.

5.5 Experimental Results

5.5.1 Experimental environment

We evaluated the performance of our compressed page cache with the Tomcat [39] web server and the SPECWeb 2005 [9] benchmark. The SPECWeb 2005 benchmark is divided into three logical components that run on different servers interconnected by a gigabit ethernet switch. The first component is the distributed client emulator that runs on a group of OpenPower 720 servers. The second is the web server that runs on a JS21 blade with two dual core Power970, 8GB of RAM, two 60GB SCSI drives and two ethernet gigabit links connected to the main switch. Finally, the third component is the database emulator (BESIM) that runs on an OpenPower 710. The JS21 server runs a 2.6.24 Linux kernel augmented with the compressed page cache (CPC), while all the other servers run a Linux distribution with a standard 2.6.9 kernel version. The SPECweb2005 benchmark and the Tomcat web server are described in more detail below.

SPECweb2005 [9] is composed of three workloads that attempt to represent three types of real web applications. The benchmark is based on a number of customers that concurrently navigate a web application switching from active to passive states and vice versa. A client in the active state performs one or more requests to the web server thereby simulating user activity, whereas a client in the passive state simulates a user think-time between active states. The time spent in passive states (or think-times) are distributed following a geometric distribution. For each workload type a Markov Chain is defined to model the client's access pattern to the web application and a Zipf distribution is used to access the file working set. The Zipf distribution used in our test is characterized with an alpha value of 0.4. SpecWeb2005 also has three other important features to mimic a real environment. Firstly, it includes the use of two connections for each client to perform parallel requests. Secondly, it simulates browser caching effects by using If-Modified-Since requests. Finally, the web server application working set size is proportional to the number of clients simulated. In this chapter we focus our attention on

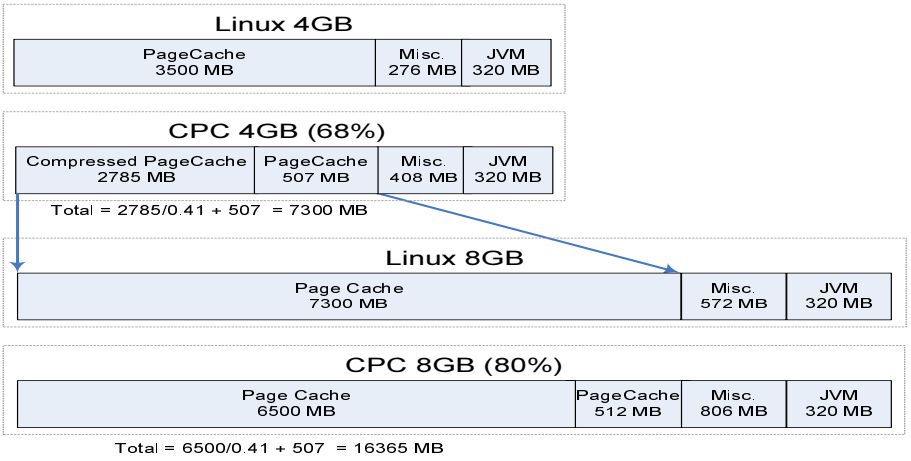


Figure 5.3 Compressed Page Cache Memory Layout

the Support workload, which is designed to simulate a vendor's support web site. Users are able to perform the typical actions in this type of web application. The two principal characteristics of the Support application are the use of only plain connections and a large working set per client, so the benchmark tends to be I/O disk intensive.

We use the Tomcat 5.5 web server with the hybrid connector to run the SPECweb2005 Support web application. The Hybrid connector is a modification of the original Coyote threading model, although both connectors share most of the code (like parsing HTTP request code or the pool of threads). The Hybrid connector follows the architecture design explained in Section 4.

5.5.2 SPECWeb2005 benchmark

In this section we evaluate the performance of the Tomcat web server running on top of the CPC vs. running the web server on top of a plain Linux kernel by using the SPECWeb 2005 Support application. The original working set of the SPECWeb Support is generated with random data, and it is therefore incompressible. In order to evaluate the benefits of our compression approach, we replaced the content of the original working set with the content of some files from the Silesia corpus [75], which is intended to represent the current content of common diskfiles. This dataset has an average compression factor of 41% with the LZO compression algorithm that is between the ranges that some papers forecast for in memory data [83], [57] and [5]. We chose the LZO algorithm because it has a good compression ratio with file backed data and is one of the fastest available compressors.

We compared the performance of four deferents configurations. Firstly, we ran the web server benchmark with the standard 2.6 Linux kernel configured with 4GB and

Name	RAM	CPC memory	PageCache Size
Linux 4GB	4GB	0%	3400MB
Linux 8GB	8GB	0%	7300MB
CPC 4GB	4GB	68%	7300MB[1]
CPC 8GB	8GB	80%	16300MB[1]

Table 5.1 Summary of configurations evaluated. [1]with a data compression factor of 41%

8GB of physical RAM. The results of both configurations were used as a bottom and upper baseline result to compare the performance obtained using the compressed page cache (CPC) on a system with 4GB of physical RAM, but configured with a page cache size equal to the 8GB configuration due to the compression effect. Finally we also ran the benchmark with the CPC and 8GB of physical RAM to prove it's scalability. The experiments are summarized in Table 5.1. As we can observe, the two plain Linux configurations with 4GB and 8GB of physical RAM have a page cache of size 3600MB and 7300MB respectively. The remaining RAM was used as a heap by the Java virtual machine that runs the Tomcat server and for Linux internal purpose, like network buffers and other non swappable slab caches. In Figure 5.3 we can see a diagram detailing the memory layout of the four configurations evaluated. The region labeled "misc" includes network buffers, a minimum pool of free pages, the array of struct pages and other non swappable memory. The region labeled JVM is the memory utilized by the Java Virtual Machine that runs the Tomcat web server. In order to isolate the effects of the CPC on the performance of the page cache we configured the system to be without a swap partition, thus the anonymous memory can not be swapped out and reclaimed. We used 68% of the memory to store compressed data on the CPC 4GB configuration in order to have a page cache as large as the plain Linux 8GB configuration. With a data compression factor of 41%, the CPC 4GB configuration with 68% of memory dedicated to store compressed data results in 2785MB of compressed data plus 507MB of uncompressed memory that adds up to a page cache size of 7300MB like the plain Linux 8GB configuration. Although the page cache size of the evaluated configurations are large, they are unable to cache all of the working set of the SPECWeb Support Workload which is considerably larger. In this benchmark the working set size is proportional to the number of clients and goes from 17GB for 1000 concurrent clients to 37.4GB for 2200 concurrent clients.

5.5.3 Performance results

In the first set of experiments we ran a total of thirteen tests on each configuration, varying the intensity of the load from 1000 to 2200 concurrent clients in increments of 100. For each test we captured a set of performance parameters returned by the benchmark client; such as the obtained throughput and the response time, as well as a number of system metrics returned by the vmstat tool and the CPC code, e.g. disk bandwidth usage, CPU utilization, page cache size and compression/decompression times. In the second set of experiments we chose the load with the best throughput from the previous experiments, for each of the CPC configurations, and then varied the percentage of memory dedicated to store compressed data from 10% to 80% in increments of 10 points.

Compressed Page Cache vs Plain Linux Kernel

In Figure 5.4 we see the throughput of all the configurations evaluated. As we can observe, all the configurations have a similar behavior with two different phases. In the first phase they increased their throughput linearly with the number of concurrent clients (or load). At a certain point the server stopped increasing its throughput and entered the second phase which is characterized by a softly decreasing throughput as the load grows. The main difference between the configurations is the point when they get saturated i.e. the change between the first and the second phase. Figure 5.5 is complementary to Figure 5.4 and shows how the response time quickly grows when a configuration reaches its saturation point. This is normal behavior observed in web servers performance [27].

As we can see in Figure 5.4 the configuration where the standard Linux kernel has 4GB of RAM was the first to reach the saturation point with a load of 1400 concurrent clients. This low throughput can be explained by looking at Figure 5.7 where we can check that the disk bandwidth utilization at this point is at its maximum (44MB/s). We verified that the bottleneck was the disk bandwidth by checking the network and CPU resource usage. The standard Linux kernel configured with 8GB of RAM had a larger page cache size than the 4GB configuration as shown in Figure 5.6. The main effect of a larger page cache is the reduction of disk accesses, saving disk bandwidth per client, delaying the saturation point and achieving better overall performance. Figure 5.4 shows how the 8GB configuration was able to reach its maximum throughput with 1700 concurrent clients.

The CPC configuration with 4GB of physical RAM performed somewhere between the other two setups. It obtained a noticeably better throughput than the 4GB configuration, but was unable to reach the levels of performance of the 8GB configuration despite the fact that both configurations had the same page cache size of 7300MB as we see in Figure 5.6. This result is explained by the disk bandwidth data plotted in

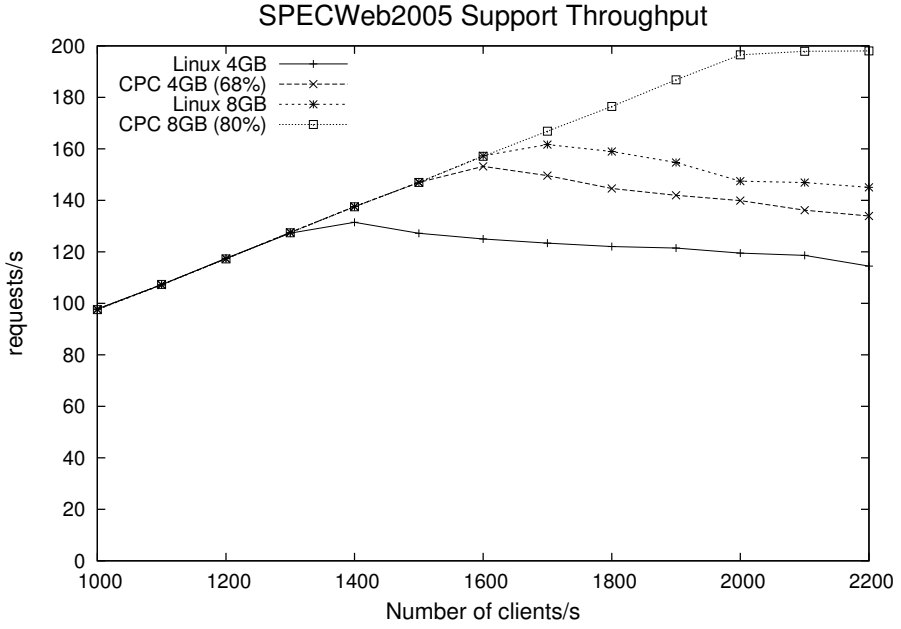


Figure 5.4 Throughput comparison

Figure 5.7 that shows how the CPC 4GB configuration was unable to exceed 41MB/s while the two configurations without memory compression reached 44MB/s. To study this effect in more detail, we fixed the load at 1700 concurrent clients and varied the percentage of memory dedicated to store compressed data from 0% to 80% in increments of 10. The throughput and disk bandwidth usage are depicted in Figure 5.8. We see the tendency for the throughput to increase as the percentage of memory being compressed increases because we need to perform less disk accesses. In contrast, the disk bandwidth utilization decreased softly, but sufficiently to prevent the CPC 4GB configuration from performing as well as the plain Linux 8GB configuration. This continuous decrease in disk performance as the percentage of compressed memory increases can not be explained as a CPU shortage because in Figure 5.9 we can see there is a CPU utilization of less than 50%. However, this behavior can be explained by how the Linux memory reclamation code is triggered.

In Linux the memory can be reclaimed through the kswapd daemon or directly by an application. In the first case, the kswapd daemon is woken up periodically and if the free memory is below a predefined threshold it starts the reclamation procedure. In the second case, the application performs a new memory allocation and if the memory is below a predefined threshold it starts the reclamation procedure. In the plain Linux kernel, executing the reclamation code has no effect on performance, but with the CPC, the discarded pages have to be compressed, and so the reclamation procedure is much slower. This fact slows down the process of obtaining new pages to perform the required

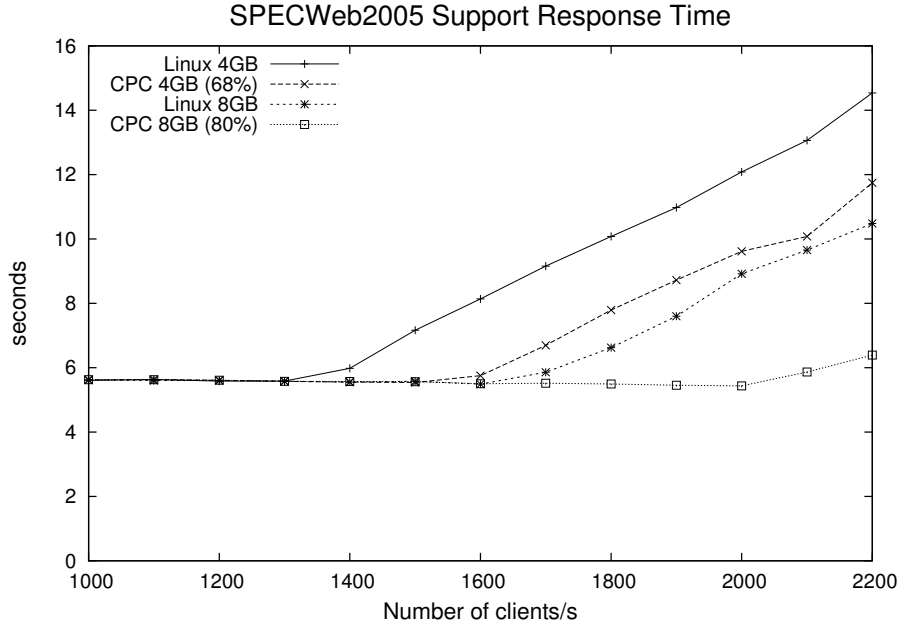


Figure 5.5 Response Time comparison

disk operations and is the cause of the lower disk read performance. We think that this problem can be solved by incrementing the memory threshold of the kswapd daemon that calls the reclamation code and launching one kswapd daemon per CPU instead of one kswapd daemon per node, thus reducing the number of times that the application triggers the direct memory reclamation procedure.

The CPU utilization results related to Figure 5.8 can be observed in Figure 5.9. As we expected, the throughput increased as the percentage of memory dedicated to store compressed data increased; more data can be cached, until a saturation point is reached, where the available uncompressed memory is so small that the system is exhausted. In Figure 5.9, the PageCache key shows the size of the CPC (the sum of both compressed and uncompressed page frames), the User key reflects the CPU time spent in the Java virtual machine, the System key shows the CPU spent on system calls, and the Compression and Decompression keys the amount of CPU used for compressing and decompressing data respectively. Figures 5.8 and 5.9 show how the CPU spent in User and System grew proportionally to the throughput obtained. In contrast, the CPU time dedicated to compressing data had a large impact as soon as we dedicated some memory to store compressed data. From that point on, the CPU spent in compression grew proportionally to the number of decompressions, which is proportional to the number of cache hits, and also grew as the compressed page cache size grew. Two key factors explain the big differences between the compression and decompression times. firstly, the compression time of a page frame is double its decompression time and secondly, we compress all

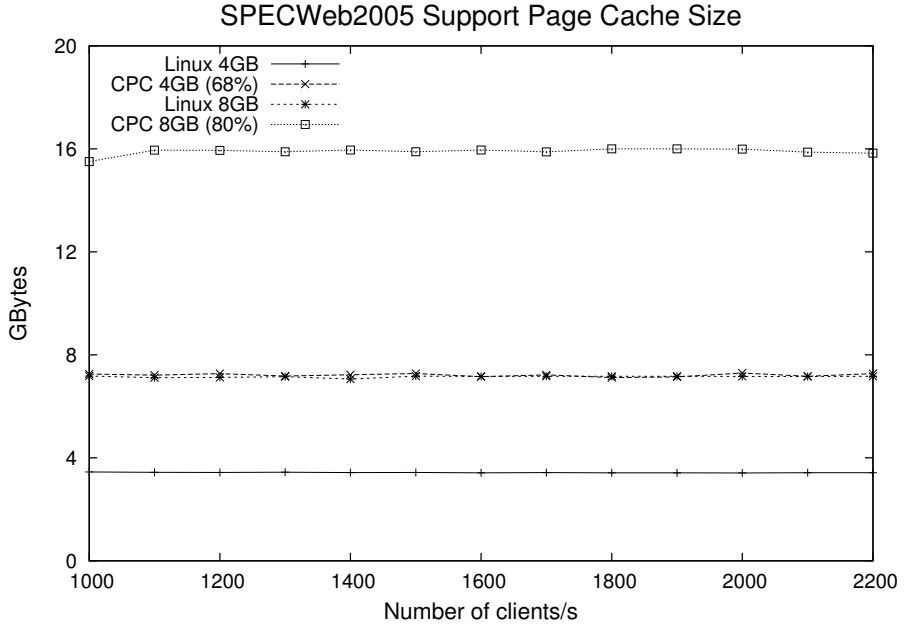


Figure 5.6 Linux Page Cache Size

the page frames when they are reclaimed, so that all the data that is read from the disk or from the compressed page cache is compressed sooner or later. In contrast, we only decompress a page when a page cache lookup has a hit; that is , for example, 40% of the time with 70% of memory dedicated to store compressed data.

Compressed Page Cache Scalability

In Figure 5.11 we can see the detailed CPU usage of the CPC 8GB configuration with a constant load of 2100 concurrent clients. As with Figure 5.9, which shows the CPU usage for the CPC 4GB configuration, we can observe how the throughput increased as the percentage of compressed memory increased and thus the PageCache size also increased. In this case, we reached the maximum throughput when 80% of the memory was dedicated to store compressed data instead of the 70% used in the CPC 4GB configuration. Figure 5.3 shows how both configurations have an uncompressed page cache of 512MB. Below this minimum size, the two configurations started to degrade their throughput due to the memory shortage. In Figure 5.10 we can observe how the throughput increased and the disk bandwidth decreased considerably as the percentage of compressed memory rose up to 85%, when the throughput started to decrease. These figures show the good scalability that our CPC has in function of the percentage of the compressed memory used. Figure 5.12 shows the trade-off between the increase in PageCache size and the computational power required to achieve it with the Support workload. As we can see,

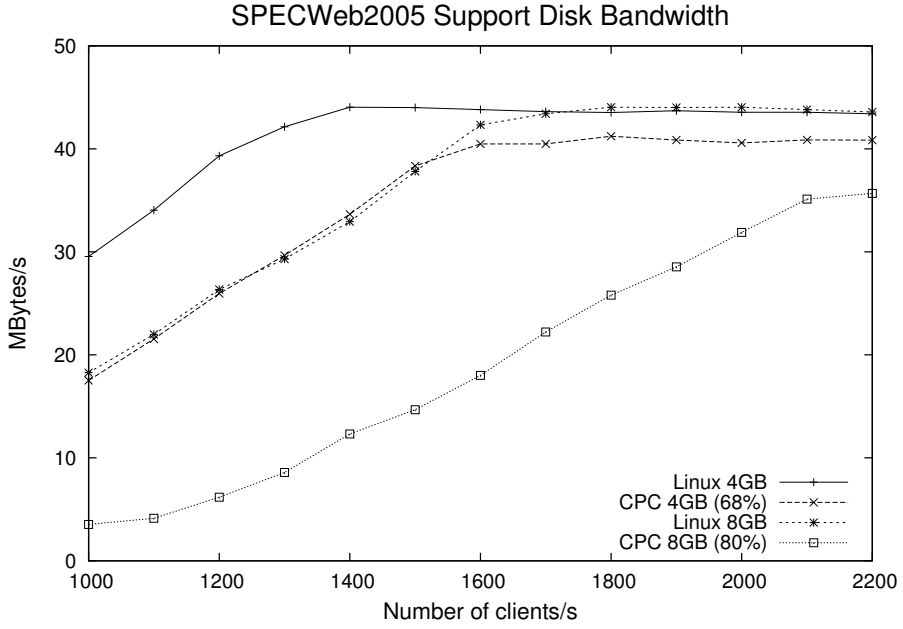


Figure 5.7 Disk Bandwidth Utilization

the processing time required and the increment of the PageCache size are proportional. We can also observe that the sum of CPU time for Compression and Decompression tasks range from 11% to 26% (with a page PageCache size increment of 18% to 120% respectively). The ability of the CPC to double the PageCache size from less than 8Bytes to slightly more than 16GBytes shown in Figures 5.10 and 5.12, produces a remarkable increase in throughput for the CPC 8GB configurations, which is depicted in Figure 5.4.

5.5.4 Synchronous vs asynchronous CPC

The implementation of the CPC used to evaluate the performance of a web server in the last section was designed to run the compression and decompression tasks synchronously in the current thread. This design can take advantage of a multiprocessor system because the Linux kernel and the evaluated web server running the SPECWeb2005 workload are highly threaded. In contrast, if an application were single threaded, with this design, it would be more difficult to fully exploit the power of larger multiprocessors systems. To solve this problem and allow the execution of compression / decompression tasks in specialized compression hardware or dedicated cores of a heterogeneous multiprocessor system like the CBE [87], we implemented in the Linux kernel a mechanism to execute these tasks asynchronously. Our framework runs on top of the work-queues facility provided by the standard Linux kernel. These work-queues have a dedicated thread for each CPU that processes the tasks enqueued. Our implementation

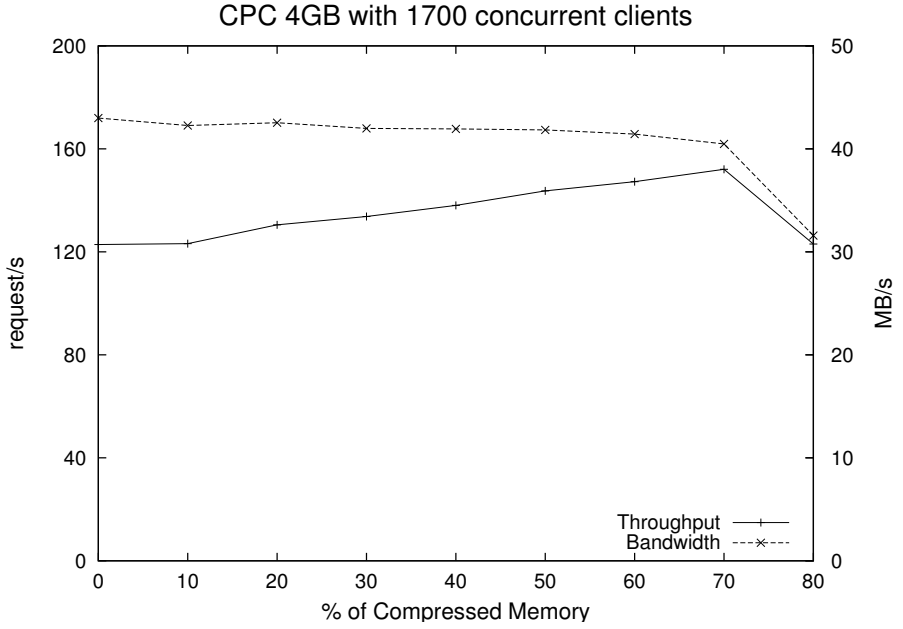


Figure 5.8 Throughput and Disk Bandwidth Trend

allows the kswapd daemon to send a batch of compression tasks to multiple work-queues increasing the processing parallelism. The decompression tasks are also enqueued to dedicated work-queues, but in this case we can not send multiple decompression tasks because the decompressions are always triggered one to one. We have evaluated and compared the asynchronous mechanism with the CPC 8GB configuration using the same parameters and the results did not show any noticeable degradation of the performance of the asynchronous CPC. Despite the overhead of enqueue tasks and context switches, the performance of both mechanisms are on a par because this overhead is small compared with the time required to perform compression and decompression tasks. With these results, we can predict that the performance of the CPC running on top of specialized hardware and heterogeneous multiprocessors are both feasible and promising. We think that our software approach can be improved by using this new kind of hardware resource to create hybrid memory compression systems which combines the flexibility of the software implementations and the performance of the specialized hardware solutions. In the next section we describe how we modified our asynchronous framework to run the CPC on the Cell processor.

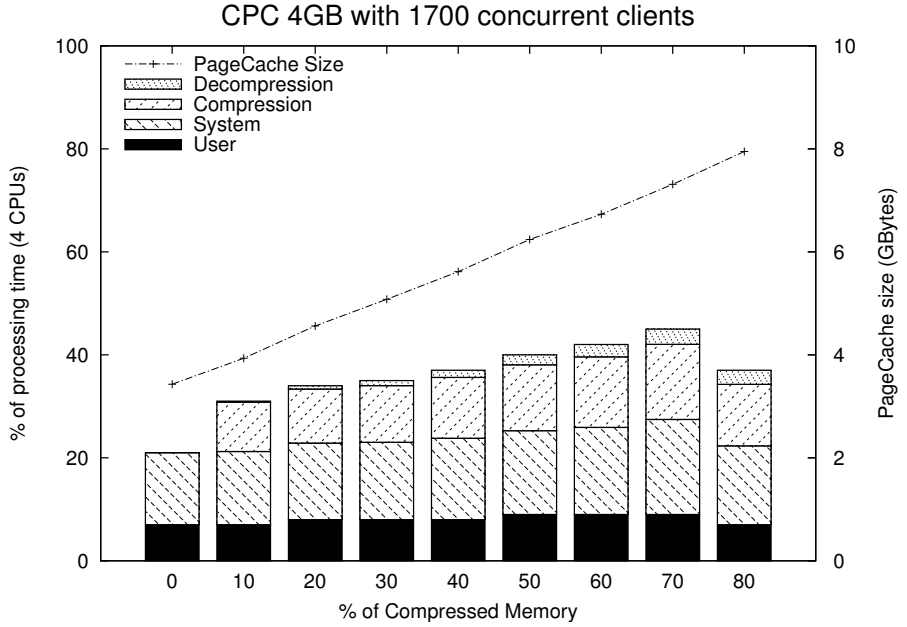


Figure 5.9 Detailed CPU usage

5.6 Cell Implementation of the CPC

As we have show in the previous sections, the compression and decompression of memory page frames is a computationally intensive operation. These tasks can be offloaded in heterogeneous architectures such as the provided by the Cell processor, executing the compression and decompression code in the specialized SPEs. With this configuration we can mitigate the load of the main processors that can focus on the execution of the user applications. However, the use of SPEs to offload the compression task presents some problems. Currently, there is no standard mechanism or framework to allow the use of the SPEs directly from the Linux kernel space. Moreover, the LZO code is highly optimized to run on modern superscalar processors with state of the art branch-prediction hardware, which is not the case of the SPE of the Cell processor.

The rest of this chapter is organized as follows: Section 5.6.1 describes the general characteristics of the Cell processor. In section 5.6.2 we introduce the Linux crypto API and the required extensions to support asynchronous compression/decompression operations. Section 5.6.2 presents a experimental framework designed to offload code from inside the kernel. Finally in section 5.6.4 we evaluate the performance of our vectorized LZO compression algorithm.

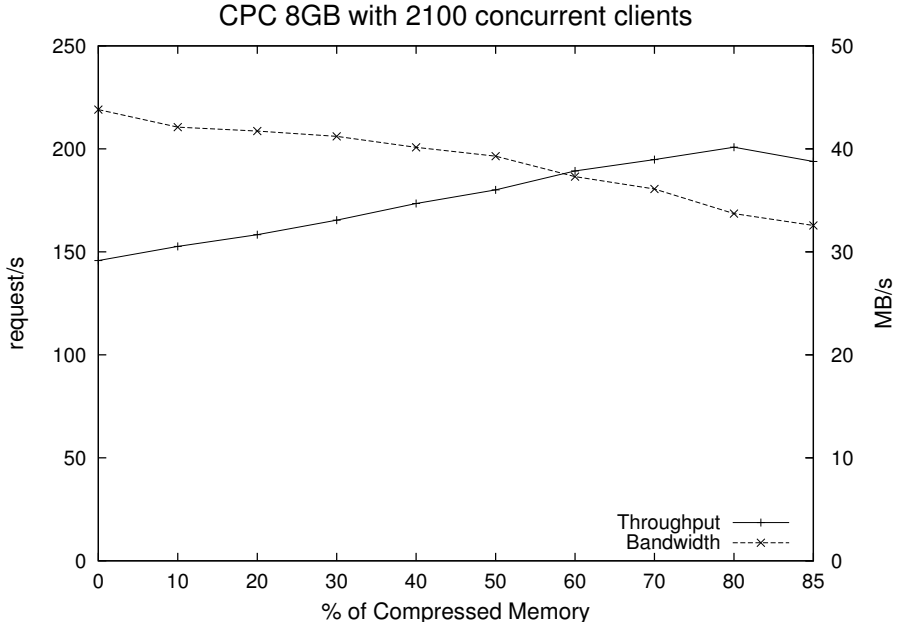


Figure 5.10 Throughput and Disk Bandwidth Trend

5.6.1 The Cell Broadband Engine Architecture

The Cell Broadband Engine Architecture (CBEA) [56] is a single chip heterogeneous multiprocessor. The design goals of the Cell processor were to address the fundamental challenges facing modern microprocessor development: high memory latencies and on-core power dissipation. Until now, microprocessors have achieved performance improvements through higher clock frequencies and deeper pipelines, but the fundamental problem that current processors face is the memory wall [89]. On modern processors significant amounts of time are spent waiting in memory stall, due to the large difference between the processor and the memory speed. Large memory latencies make it difficult to obtain further performance gains with traditional processor designs based on hardware caches. The Cell processor approaches this problem in a different way, providing a heterogeneous processor with explicit memory management. This approach potentially improves the throughput of the processor, but also increase the effort to efficiently implement a program.

Figure 5.13 shows the three basic components of the Cell processor. First, the PowerPC Processor Element (PPE), which is primarily intended to manage global resources. Second, the Synergistic Processing Elements (SPE) that are specialized vectorial processors. Finally, the communication between the PPE, the SPEs, main memory, and external devices is realized through an Element Interconnect Bus (EIB).

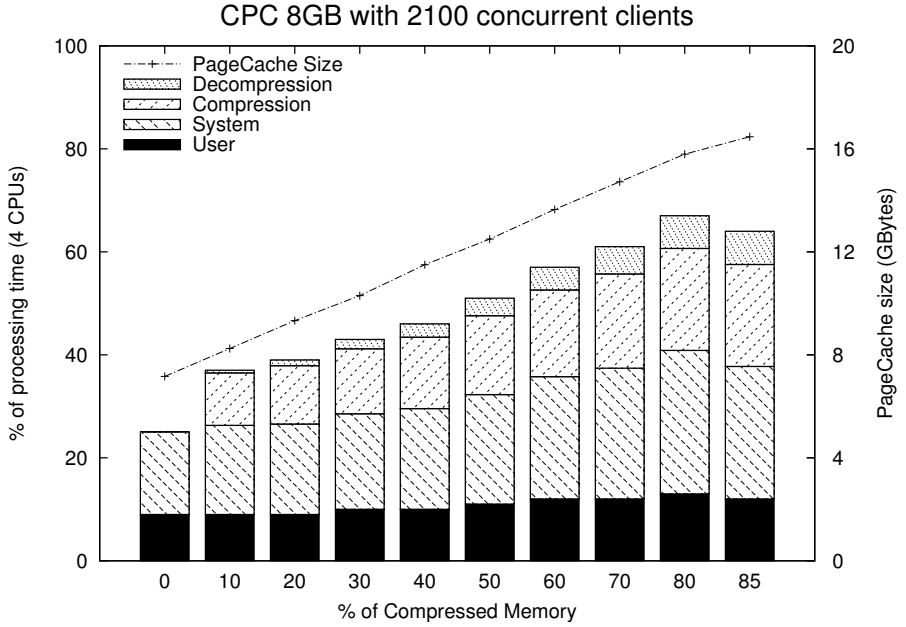


Figure 5.11 Detailed CPU usage

The Power Processor Element

The PPE is the main Cell processor designed to be power efficient and manage all other system peripherals and processor cores. It is a dual-issue in-order execution design. It provides two hardware threads that can be executed simultaneously. The PPE is a 64 bits processor with a vectorial unit (VMX) that has the usual cache hierarchy with a 32-KB first-level (L1) instruction and data cache and a 512-KB second level (L2) cache used to hide memory latencies. The PPE is in charge of executing the Operating System, as well as distributing the load between all the SPE units. The communication between the PPE and the SPE is done through shared memory regions or through direct mail boxes provided by the SPE units.

The Synergistic Processing Element

SPEs are specialized vectorial units with an instruction set similar (but not compatible) to the PPE VMX. The main difference is found in the memory hierarchy, which is divided into three levels: the main memory, the local stores and large unified register files. The large unified register file makes it possible to hold the majority of operands directly inside the CPU core without having to spill values onto the stack. A 256Kb local store is used to store the SPE code and temporary data. SPEs can perform asynchronous DMA transfers between their local stores and main memory

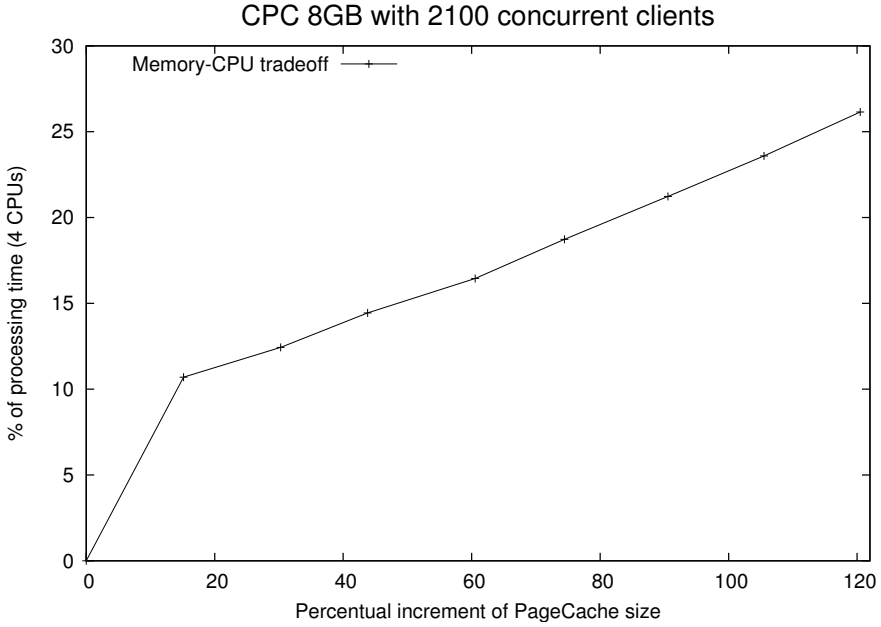


Figure 5.12 Memory and CPU trade-off

SPEs are designed to execute regular computationally intensive programs rather than general purpose software. This allows the system to hide memory latencies without having to employ complex hardware mechanisms such as branch-prediction, out-of-order execution, and deep pipelining, often used in superscalar processor cores. This permits the reduction of the hardware required to obtain a high throughput and hardware utilization on regular programs, but at the cost of requiring a considerable effort in order to optimize non-regular programs in such a wat that an acceptable performance is obtained. The simple design of the SPE cores makes it possible to pack together up to eight SPU in a single multi-core-die.

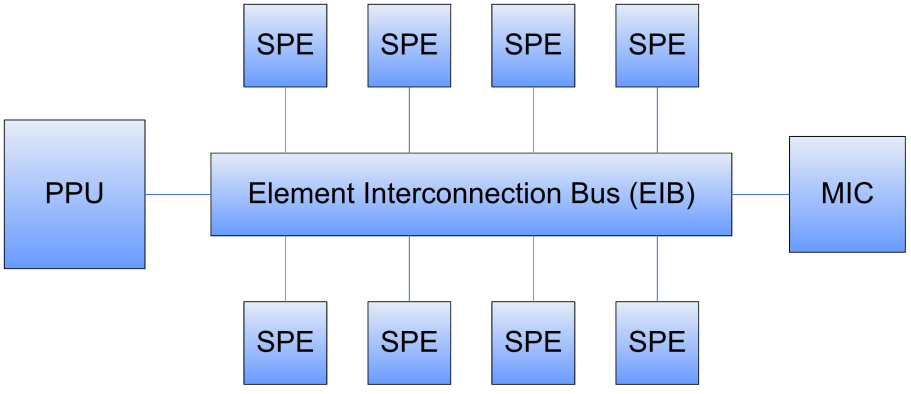


Figure 5.13 The Cell Broadband Engine Architecture

The Element Interconnect Bus

Communication between SPEs and the Element Interconnect Bus (EIB) is realized through the SPEs Memory Flow Controller (MFC). The MFC of each SPE can enqueue up to 16 DMA commands, which implies that the whole system can process more than 100 DMAs simultaneously. It also provides memory mapped I/O registers (MMIO) and channels to monitor DMA commands, SPU events and facilitate interprocess communication via mailboxes and signal-notification. Mailboxes are a set of queues that support exchanges of 32-bit messages between an SPE and other devices. Two one-entry mailbox queues are provided for sending messages from the SPE. The EIB is a 4-ring structure (2 clockwise, 2 counterwise) for data, and a tree structure for commands with an internal bandwidth of 96 Bytes per cycle. The EIB has two external interfaces, the MIC, which is the interface between main memory and EIB and the BEI, which allows data transfer between EIB and I/O devices.

5.6.2 Linux Crypto API

The Linux crypto API is used to publish cryptographic, compression and digest algorithms in a unified way to in-kernel users like IPsec, wireless drivers or file systems drivers. As we can see in Figure 5.14, all the encryption, compression and hashing algorithm implement the synchronous API. This API is the most frequently used inside the kernel, as its use is easy and natural. In contrast, the asynchronous API, built on top of the synchronous one, provides more advanced functionalities and was specifically designed to take advantage of specialized encryption hardware. This API provides a way to asynchronously send a batch of request, thus improving the performance and use of specialized hardware such as cryptographic processors.

The original Linux asynchronous crypto API does not support compression or hashing algorithm, and so our first step was to extend this asynchronous API so that it was to be able to deal with a compression algorithm. The next step was to ensure that we could call the Kspu framework form our asynchronous compression API, and this is explained in the next section.

5.6.3 Kspu Framework

The Kspu framework is an experimental design by Sebastian Siewior [76] which allows the execution of arbitrary code on the SPUs from inside the kernel. The primary objective is to be able to offload cryptographic algorithms from the Crypto API, but the framework can also be used to execute arbitrary kernel code. The integration of

the asynchronous compression API and the Kspu framework did not imply any major problems, as the steps needed to encrypt or compress a bloc of data are very similar. Conceptually, this framework permits the offload of specific kernel functions to the SPEs.

As an example, an in-kernel user enqueues a request using the asynchronous compression API. This API enqueues the request in a dedicated queue of the Kspu framework. When the SPU is ready, the request is copied to the SPU and the buffers needed are fetched from the SPU. When all the data required is in the SPU, the compression algorithm is executed and the resulting data copied back to main memory. Finally, the user handler is called to notify the finalization of the process.

Currently, the framework has some limitations such as the impossibility of running kernel code on multiples SPUs at the same time. This limitation is not very relevant to our experiments because, as we will show in the next section, with our optimized LZO version we have enough power to deal with all compression and decompression required by our CPC. Further details of the Kspu framework can be found in [76].

5.6.4 LZO vectorization

The initial port of the LZO algorithm to the SPU was a bit disappointing, as we only obtained one quarter of the performance required to offload the compression task of our experiments with the SPECweb2005 benchmark, explained in Section 5.5.3. The poor performance of the original LZO code can be explained by the specialized characteristics of the SPU cores. These cores do not have branch prediction hardware and mispredicted branches incur high penalties. Moreover, the original LZO code has many branches that can not be accurately predicted at compilation time. The original code is completely scalar, so it cannot capitalize on the vectorial nature of the SPU cores.

To improve the performance of the code we had to completely rewrite the compression algorithm to avoid as far as possible the most frequent branch instructions, and also to vectorize the most time consuming tasks in the algorithm. Some control dependencies were changed to data dependencies. In addition, the string hash calculation and the string comparison were vectorized to speed up the algorithm. With this optimizations we greatly improved the performance of the original LZO code as we can see in Figure 5.15. This figure shows the performance of the LZO code with a common working set. This working set is composed of the first 100MB of the Wikipedia. The figure shows the performance of the original LZO code on three different CPUs at the same clock frequency (3200 Mhz): a Pentium 4, the PPU core of the Cell processor and the SPU core of the Cell processor. Finally, the figure shows the performance of the vectorized LZO code on the SPU of the Cell processor. As we can see the vectorized code is three times faster than the original

LZO code on the SPU processor and 20% faster than the original code on a Pentium 4 processor. If we take into account that the cell processor has eight SPU, the aggregated compression bandwidth is nearly ten times larger than that obtained with a Pentium 4 processor.

5.7 Summary

We implemented on the Linux OS a main memory compression system that takes advantage of the full power of current multiprocessors architectures. We evaluated its performance with a highly threaded web server running the realistic SPECWeb2005 benchmark and obtained positive results such as a 30% throughput improvement and a 70% reduction in the disk bandwidth usage. Our CPC implementation allowed us to maximize the utilization of multi-core and multiprocessor systems by memory-bounded applications, interchanging CPU cycles with memory space in a flexible manner.

With the results obtained from our asynchronous implementation we anticipate that this technology will have a big impact, when used in conjunction with new multiprocessor and multi-core technologies such as the Niagara [59] and CELL [87] processors, which have the power to accelerate the compression and decompression tasks, opening up the performance improvement to a wider set of applications bounded by the memory size or disk I/O bandwidth. Our results show that with our vectorized LZO algorithm just one SPU of the CELL processor is enough to satisfy the compression and decompression required in the experiments of Section 5.5.3 Finally, it is worth noting that our CPC implementation can utilize almost all of the physical memory to store compressed data and can improve the overall performance of the system by a large margin.

In the future, we will study the benefits of sending anonymous pages to disk in a compressed form to effectively increase the bandwidth of the disk by the data compression factor. We think that this approach can also be applied at the filesystem level and it can make an important contribution to the task of reducing the large number of compressions that are now required. Our final objective is to explore the integration of the memory compression at the filesystem and kernel level in order to produce a synergistic effect that dramatically reduces the required bandwidth in bringing up data to the main memory, as well as ensuring that compressions only apply to data that have been modified.

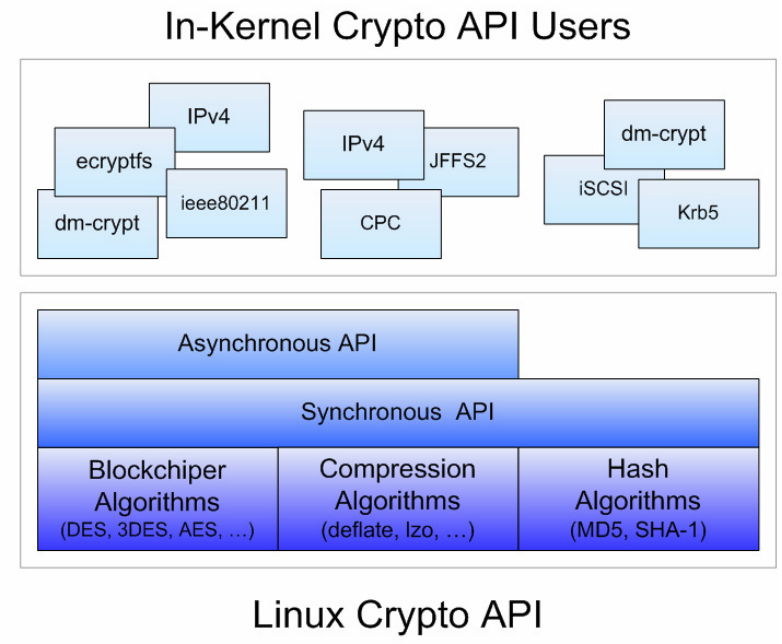


Figure 5.14 Linux Crypto API extended with asynchronous compression

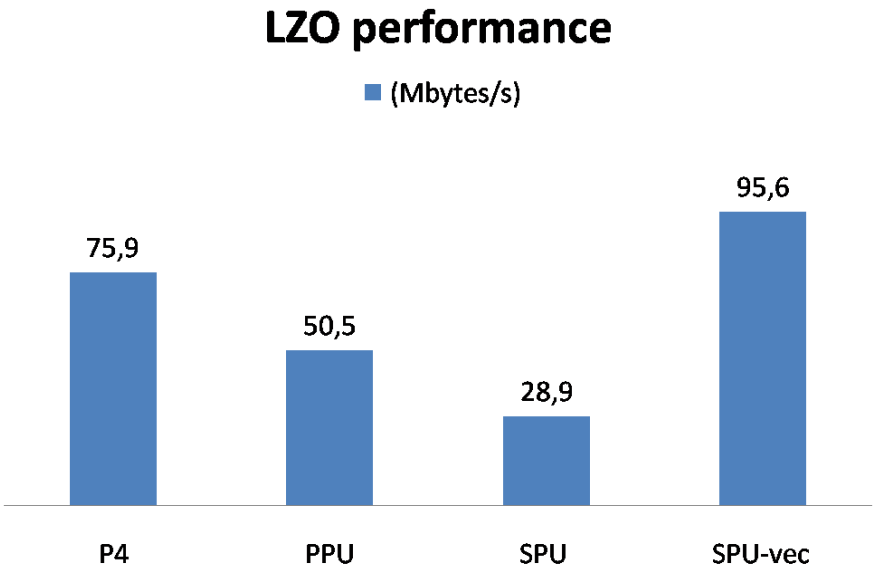


Figure 5.15 LZO performance on several processors

Chapter 6

Conclusions

In this thesis we studied the limitations and drawbacks of current web server systems. We identified two main issues that prevent web servers from acheiving good resource utilization: the classical multi-threaded architecture and the memory-bounded or disk-bounded nature of some web application workloads. To overcome these issues we proposed two novel techniques: the Hybrid architecture, which augments the classical multi-threaded architecture with a connection management system borrowed from the event-driven architecture, and a main memory compression system, which alleviates memory bounded applications by using the available processing resources present on current multi-core systems. We thoroughly evaluated both techniques to prove their feasibility on current web server systems.

Web Server System Characterization

To be able to identify the previously mentioned issues, we characterized both the multi-threaded architecture and the event-driven architecture in many environments, including CPU-bounded and network-bounded scenarios. Then, we implemented with the NIO (Non-blocking I/O) API a simple event-driven web server to test the scalability and suitability of the Java platform as a high performance web server. We performed an in-depth study of the behavior of the multi-threaded Tomcat web server under both static and dynamic workloads to identify the problems that the multi-threaded architecture experienced in its attempts to effectively manage persistent connections.

Our study of the Secure Socket Layer protocol identified the SSL handshake as a computational expensive process, which can easily saturate a current multi-threaded web server. We extended our study to include a complete characterization of Tomcat application server scalability when executing the RUBiS benchmark using SSL. This characterization includes a vertical scalability measurement, which confirmed that running with more processors makes the server able to handle more clients, but that does not represent an optimal solution. We analyzed the causes of server saturation when running with different numbers of processors using a performance analysis framework. This framework allows a fine-grained analysis of dynamic web applications by considering all levels involved in their execution. Our analysis revealed that the processor is a bottleneck for Tomcat performance on secure environments and it suggest that upgrading the system by adding more processors will improve the server scalability. The results obtained in this work demonstrate the value of incorporating some kind of overload control mechanism in the Tomcat system, to avoid the throughput degradation produced by the massive arrival of full SSL connections, because the multi-threaded architecture is not well designed to handle these kinds of situations.

We identified the configuration complexity of the multi-threaded architecture as a key problem that hinders a web server from obtaining optimal performance. The inherit difficulty of configuring a multi-threaded web server can be explained by the introduction of the connection persistence in the HTTP 1.1 protocol, which forces the introduction of a keep-alive timeout to allow worker threads to serve different active connections. This mechanism does not scale well when the load of the web server increases or changes, making the configuration of a multi-threaded web server a challenging task. We studied and solved the identified problems of the multi-threaded architecture in Chapter 4, with the introduction of our Hybrid architecture.

Hybrid Web Server Architecture

In Chapter 4, we showed how a web server with a Hybrid architecture which combines the best of a multi-threaded design with the best of an event-driven model can be used to obtain a high performance web server, especially when the throughput is measured in successfully completed user sessions per second. The proposed implementation into the Tomcat 5.5 code offers a slightly better performance than the original multi-threaded Tomcat server when it is tested with a static content application, and a remarkable improvements in performance when it is compared to the multi-threaded architecture in a dynamic content scenario, where each user session failure can be directly related to business revenue losses. Additionally, the natural way of programming introduced by the multi-threading paradigm can be maintained for most of the web container code. The Hybrid architecture can boost the performance of web server systems, especially under session-based workloads and even more when the workloads use encryption techniques. As has been demonstrated, the benefits of this architecture are especially noticeable in the throughput of the server, in terms of individual requests as well as for user sessions, particularly when the server is overloaded. The modified Tomcat server beats the original multi-threaded Tomcat in all the performance parameters studied. We also studied the complexity of optimally configuring a multi-threaded web server and we demonstrated that the Hybrid architecture can consolidate different types of workload on the same server without the need for manual tuning or configuring. From our results we demonstrated the inherent difficulty of optimally configuring a multi-threaded web server due to the interaction between its internal configuration parameters and the external workload characteristics. Furthermore, we demostrated how the Hybrid architecture can dramatically reduce the configuration complexity and increase the dynamic web server's adaptability to workload intensity and workload type changes, thus making it a good solution for environments like web farms, where the servers need to deal with different

types of workloads at the same time. We also showhed that the Hybrid connector performs better than the multi-threaded one on the two CPU intensive workloads (50% and 30% throughput improvement for the Banking and E-commerce workloads respectively) and offers the same performance on the I/O disk intensive workload.

Main Memory Compression

In the process of studying the multi-threaded and Hybrid web server architecture we found that some web applications workloads are memory-bounded or disk-bounded, and so they are unable to make the most of all the CPU power available on current multi-core hardware. To alleviate this problems we proposed a main memory compression technique implemented in the operating system. Although main memory compression is not a new technique per-se, its use in a multi-core environment running heavily multi-threaded applications like a web server introduces new challenges in the technique, such as scalability issues and the trade-off between the compressed memory size and the computational power required to achieve it. We demostrated that the use of memory compression techniques to improve highly-threaded memory-bounded applications is feasible nowadays, thanks to the proliferation of multi-core and multi-processor platforms, which are the commodity hardware of current web server systems.

We implemented a main memory compression system on the Linux OS that takes advantage of the full power of current multiprocessors architectures. We evaluated its performance with a highly threaded web server running the realistic SPECWeb2005 benchmark and obtained positive results such as a 30% throughput improvement and a 70% reduction in the disk bandwidth usage. Our Compressed Page Cache (CPC) implementation allows us to maximize the utilization of multi-core and multiprocessor systems by memory-bounded applications, interchanging CPU cycles with memory space in a flexible manner.

With the results obtained from our asynchronous implementation, we anticipate that this technology will have a big impact when used in conjunction with new multiprocessor and multi-core technologies such as the Niagara [59] and CELL [87] processors, which have the power to accelerate the compression and decompression tasks, and thus open up the performance improvement to a wider set of applications bounded by the memory size or disk I/O bandwidth. Our results show that our CPC implementation can utilize almost all of the physical memory to store compressed data and improve the overall performance of the system by a large margin.

In the future, we will investigate the possibility of sending anonymous pages to disk in a compressed form in order to effectively increase the bandwidth of the disk by the

data compression factor. We think that this approach can also be applied at the filesystem level and can have a big impact on reducing the large number of compressions that are now required. Our final objective is to explore the integration of memory compression at the filesystem and kernel level so as to produce a synergistic effect that can dramatically reduce the required bandwidth while bringing up data to the main memory, as well as avoiding compressions in cases where data have not been modified.

Bibliography

[1] *Apache Software Foundation. Apache HTTPD Project. Event MPM. http://httpd.apache.org/docs/2.2/mod/event.html*. 4.5

[2] *Apache Software Foundation. Apache Portable Runtime. http://apr.apache.org/*. 2.3.3, 4.3.2

[3] *Apache Software Foundation. Tomcat APR connector. http://tomcat.apache.org/tomcat-5.5-doc/apr.html*. 4.5

[4] *Apache Software Foundation. Tomcat NIO connector. http://tomcat.apache.org/tomcat-6.0-doc/config/http.html*. 4.5

[5] B. Abali, H. Franke, D.E. Poff, R.A. Saccone, C.O. Shulz, L.M. Herger, and T.B. Smith, "Memory Expansion Technology (MXT): Software Support and Performance," IBM I. Research and Development, vol. 45, no, 2, 2001. 5.2, 5.5.2

[6] Michael J. Freedman. The Compression Cache: Virtual Memory Compression for Handheld Computers. Technical report, Parallel and Distributed Operating Systems Group, MIT Lab for Computer Science, Cambridge, 2000. 5.2

[7] *MySQL. http://www.mysql.com/*. 3.2.6

[8] S. Arramreddy, D. Har, K. Mak, et al, "IBM X-Press Memory Compression Technology Debuts in a ServerWorks NorthBridge", HOT Chips 12 Symposium, Aug. 2000. 5.2

[9] *Standard Performance Evaluation Corporation. SPECweb2005. http://www.spec.org/web2005/*. 2.4.4, 4.3.2, 4.3.2, 5.5.1

[10] *Sun Microsystems, Inc. Grizzly NIO connector. https://grizzly.dev.java.net/*. 4.5

[11] Architectural impact of secure socket layer on internet servers. In *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*, page 7, Washington, DC, USA, 2000. IEEE Computer Society. 3.2.2

[12] *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004. 3.2.6

[13] B. Abali, M. Banikazemi, X. Shen, H. Franke, D. E. Poff, and T. B. Smith. Hardware compressed main memory: Operating system support and performance evaluation. *IEEE Trans. Comput.*, 50(11):1219–1233, 2001. 5.2

[14] A. Alameldeen and D. Wood. Adaptive Cache Compression for High-Performance Processors. In *31 st Annual International Symposium on Computer Architecture*, June 2004. 5.2

[15] C. Amza, A. Chanda, E. Cecchet, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks, 2002. 2.4.3, 3.2.2, 3.2.6, 4.1.5, 4.2.3

[16] Apostolopoulos, Peris, and Saha. Transport layer security: How much does it really cost? In *INFOCOM: The Conference on Computer Communications, joint conference of the IEEE Computer and Communications Societies*, 1999. 3.2.2

[17] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998. 2.4.2, 4.1.5

[18] BEA Systems, Inc. *Achieving Scalability and High Availability for E-Business. BEA white paper. March 2003. .* 3.2.2

[19] V. Beltran, D. Carrera, J. Guitart, J. Torres, and E. Ayguadé. A hybrid web server architecture for secure e-business web applications. In *HPCC '05: Proceedings of the 1st International Conference on High Performance Computing and Communications*, pages 366–377, Sorrento, Italy, 21-22 September 2005. 1.1, 5.2

[20] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Evaluating the scalability of java event-driven web servers. In *ICPP' 04: Proceedings of the 2004 International Conference on Parallel Processing*, pages 134–142, Montreal, Quebec, Canada, 15-18 August 2004. 1.1, 3.2.2, 4.1.3

[21] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Evaluating the new java 1.4 nio api for web servers. Technical Report UPC-DAC-2004-18 / UPC-CEPBA-2004-4, Technical University of Catalonia (UPC), 2004. 3.1.5, 3.1.5, 3.1.6

[22] V. Beltran, J. Guitart, D. Carrera, J. Torres, E. Ayguadé, and J. Labarta. Performance impact of using ssl on dynamic web applications. In *Jornadas '04: Proceedings of the 15th Jornadas de Paralelismo*, pages 471–476, 15-17 September 2004. 1.1

[23] V. Beltran, J. Torres, and E. Ayguadé. Understanding tuning complexity in multithreaded and hybrid web servers. In *IPDPS '08: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, pages 56–64, Miami, FL, USA, 14-18 April 2008. IEEE Computer Society. 1.1

[24] V. Beltran, J. Torres, and E. Ayguadé. Improving web server performance through main memory compression. In *ICPADS '08: Submitted to the 14th International Conference on Parallel and Distributed Systems*, Melbourne, Victoria, Australia, December 2008. IEEE Computer Society. 1.1

[25] V. Beltran, J. Torres, and E. Ayguadé. Improving disk bandwidth-bound applications through main memory compression. In *MEDEA '07: Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications , systems and architecture*, pages 57–63, Brasov, Romania, September 2007. ACM. 1.1

[26] D. Boneh and H. Shacham. Fast variants of rsa, 2002. 3.2.2

[27] D. Carrera, V. Beltran, J. Torres, and E. Ayguadé. A hybrid web server architecture for e-commerce applications. In *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems*, pages 182–188, Fuduoka, Japan, 20-22 July 2005. IEEE Computer Society. 1.1, 4.3.2, 5.2, 5.5.3

[28] D. Carrera, J. Guitart, J. Torres, E. Ayguadé, and J. Labarta. Complete instrumentation requirements for performance analysis of web based technologies. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2003. 2.5

[29] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 246–261, 2002. 3.2.2, 3.2.6

[30] R. Cervera, T. Cortes, and Y. Becerra. Improving Application Performance through Swap Compression. In *Proceedings of the USENIX Technical Conference (Freenix track)*, 1999. 5.2

[31] H. Chen and P. Mohapatra. Session-based overload control in qos-aware web servers. In *INFOCOM*, 2002. 4.1.3

[32] C. Coarfa, P. Druschel, and D. S. Wallach. Performance analysis of tls web servers. *ACM Trans. Comput. Syst.*, 24(1):39–69, 2006. 3.2.2

[33] Computer Industry Almanac Inc. *http://c-i-a.com/*. 1.1a

[34] R. S. de Castro, A. P. do Lago, and D. D. Silva. Adaptive Compressed Caching: Design and Implementation. In *SBAC-PAD '03: Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, page 10, Washington, DC, USA, 2003. IEEE Computer Society. 5.2

[35] Y. Diao, J. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Using mimo linear control for load balancing in computing systems. In *American Control Conference, 2004. Proceedings of the 2004, vol.3*, pages 2045–2050, 30 June 2004. 2.3.3, 3.3, 4.3.1, 4.3.2

[36] T. Dierks and C. Allen. The tls protocol version 1.0, 1999. 3.2.1, 3.2.3, 4.2.2

[37] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *USENIX Winter*, pages 519–529, 1993. 5.2

[38] M. Ekman and P. Stenstrom. A Robust Main-Memory Compression Scheme. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society. 5.2

[39] A. S. Foundation. *Jakarta Project*. 3.2.6, 4.1.2, 4.2.1, 4.3.2, 5.5.1

[40] A. Goldberg, R. Buff, and A. Schmitt. Secure web server performance dramatically improved by caching ssl session keys, 1998. 3.2.2

[41] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Characterizing secure dynamic web applications scalability. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pages 108–116, Denver, CO, USA, 04-08 April 2005. 1.1, 4.2.1, 4.2.2, 4.3.1, 5.2

[42] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguade. Session-based adaptive overload control for secure dynamic web applications. *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 341–349, 14-17 June 2005. 5.2

[43] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguade. Session-based adaptive overload control for secure dynamic web applications. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 341–349, Washington, DC, USA, 2005. IEEE Computer Society. 1.1

[44] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguadé. Designing an overload control strategy for secure e-commerce applications. *Comput. Netw.*, 51(15):4492–4510, 2007. 1.1, 3.2.2

[45] J. Guitart, D. Carrera, V. Beltran, J. Torres, and E. Ayguadé. Dynamic cpu provisioning for self-managed secure web applications in smp hosting platforms. *Comput. Netw.*, 52(7):1390–1409, 2008. 1.1

[46] J. Guitart, J. Torres, E. Ayguad, and J. Labarta. Java instrumentation suite: Accurate analysis of java threaded applications, 2000. 3.2.6

[47] I. Haddad and G. Butler. Experimental studies of scalability in clustered web systems. *ipdps*, 09:185b, 2004. 3.2.2

[48] I. F. Haddad. Open-source web servers: performance on carrier-class linux platform. *Linux J.*, 2001(91):1, 2001. 3.2.2, 3.2.5

[49] S. Harizopoulos and A. Ailamaki. Affinity scheduling in staged server architectures. Technical Report CMU-CS-02-113, Carnegie Mellon University, 2002. 3.1.1

[50] R. Hitchens. *Java NIO*. O'Reilly, 2002. 2.3.3, 4.3.2

[51] J. Hu, I. Pyarali, and D. Schmidt. Applying the proactor pattern to high-performance web servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems. IASTED*, October 1998. 3.1.1

[52] J. Hu and D. Schmidt. *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. Wiley & Sons, 1999. 3.1.1

[53] IBM Corporation. WebSphere Application Server. http://www.ibm.com/websphere. 3.1.1, 3.2.2

[54] IBM Research. *Autonomic computing. See http://www.research.ibm.com/autonomic*. 4.1.7

[55] H. Jamjoom, C.-T. Chou, and K. G. Shin. The impact of concurrency gains on the analysis and control of multi-threaded internet services. In *INFOCOM 2004.*

Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, pages 827–837 vol.2, Hong Kong, China, 2004. 4.3.1

[56] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005. 5.6.1

[57] S. F. Kaplan. Compressed Caching and Modern Virtual Memory Simulation, Ph.D. Thesis, University of Texas at Austin, December 1999. 5.2, 5.5.2

[58] A. Keller and H. Ludwig. Defining and monitoring service-level agreements for dynamic e-business. In *LISA*, pages 189–204, 2002. 4.1.2

[59] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*. 5.7, 6

[60] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance (extended abstract). In *LCTES/OM*, pages 182–187, 2001. 3.1.1

[61] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. Parekh. Online response time optimization of apache web server. In *Quality of Service - IWQoS 2003: 11th International Workshop*, pages 461–478, Berkeley, CA, USA, 2003. 4.3.1

[62] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998. 2.4.1, 4.1.5, 4.2.3

[63] National Center for Supercomputing Applications. *http://en.wikipedia.org/wiki/Mosaic_web_browser/*. 1

[64] Netcraft Webserver Survey. *http://www.netcraft.com/survey/*. 1.1b

[65] One Laptop per Child Foundation. *One Laptop per Child Project. http://laptop.org/*. 5.2

[66] T. Osogami and S. Kato. Optimizing system configurations quickly by guessing at the performance. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 145–156, New York, NY, USA, 2007. ACM Press. 4.3.1

[67] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999. 1, 3.1.1, 4.1.2, 4.1.3, 4.2.1

[68] Paraver Visualization Tool. *http://www.cepba.upc.es/paraver*. 2.5

[69] Peter Lin. *So You Want High Performance (Tomcat Performance). September 2003.http://jakarta.apache.org/tomcat/articles/performance.pdf*. 3.2.2

[70] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31, Amsterdam, 1995. IOS Press. 3.2.6

[71] M. Raghavachari, D. Reimer, and R. D. Johnson. The deployer's problem: configuring application servers for performance and reliability. In *ICSE '03: Proc. of the 25th International Conference on Software Engineering*, pages 484–489, Washington, DC, USA, 2003. IEEE Computer Society. 4.3.1

[72] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000. 3.2.1

[73] Rodrigo S. de Castro. *Compressed Caching for Linux. http://linuxcompressed.sourceforge.net/*. 5.2

[74] S. Roy, R. Kumar, and M. Prvulovic. Improving System Performance with Compressed Memory. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 66, Washington, DC, USA, 2001. IEEE Computer Society. 5.2

[75] Sebastian Deorowicz. *Silesia Compression Corpus. http://www-zo.iinf.polsl.gliwice.pl/ sdeor/silesia.html*. 5.5.2

[76] Sebastian Siewior. 5.6.3

[77] H. Shacham and D. Boneh. Improving ssl handshake performance via batching. In *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*, pages 28–43, London, UK, 2001. Springer-Verlag. 3.2.2

[78] M. Sopitkamol and D. A. Menascé. A method for evaluating the impact of software configuration parameters on e-commerce sites. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 53–64, New York, NY, USA, 2005. ACM Press. 2.3.3, 3.3, 4.3.1, 4.3.2

[79] Sun Microsystems, Inc. Java 2 Platform, Enterprise Edition (J2EE). http://java.sun.com/j2ee. 3.1.1

[80] Sun Microsystems, Inc. *New I/O APIs. 2002. http://java.sun.com/j2se/1.4.2/docs/guide/nio*. 3.1.1

[81] The Apache Software Foundation. *Apache HTTP Server Project. http://httpd.apache.org*. 3.1.1, 4.2.1

[82] R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy. Pinnacle: IBM MXT in a Memory Controller Chip. *IEEE Micro*, 21(2):56–68, 2001. 5.2

[83] I. C. Tuduce and T. R. Gross. Adaptive Main Memory Compression. In *USENIX Annual Technical Conference, General Track*, pages 237–250, 2005. 5.1, 5.2, 5.5.2

[84] M. Welsh. *NBIO: Nonblocking I/O for Java. http://www.eecs.harvard.edu/ mdw/proj/java-nbio*. 3.1.1

[85] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001. 1, 3.1.1, 4.1.2, 4.1.4, 4.2.1

[86] M. Welsh, S. Gribble, E. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, UC Berkeley, April 2000. 3.1.1

[87] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press. 5.5.4, 5.7, 6

[88] P. R. Wilson. Operating System Support for Small Objects. In *Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, CA, October 1991. IEEE Press. 5.2

[89] W. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. Technical report, Charlottesville, VA, USA, 1994. 5.6.1

[90] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW '04: Proceedings of the*

13th international conference on World Wide Web, pages 287–296, New York, NY, USA, 2004. ACM Press. 4.3.1

[91] K. S. Yim, J. Kim, and K. Koh. Performance Analysis of On-Chip Cache and Main Memory Compression Systems for High-End Parallel Computers. In *PDPTA*, pages 469–475, 2004. 5.2

[92] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003. 3.1.1

[93] W. Zheng, R. Bianchini, and Nguyen. Automatic configuration of internet services. In *Proceedings of the 2007 Conference on Eurosys*, pages 219–229, New York, NY, USA, 2007. EuroSys '07. ACM Press. 4.3.1